1 Fragen zu Theoretischer Informatik 1

1.1 Laufzeitanalyse

- **Lauzeitverhalten:** Existiert der Limes der Folge $\frac{f(n)}{g(n)}$ mit $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$, dann gilt:

$$\begin{aligned} c = 0 &\Rightarrow f = o(g) \\ 0 < c < \infty &\Rightarrow f = \Theta(g) \\ c = \infty &\Rightarrow f = w(g) \\ 0 \leq c < \infty &\Rightarrow f = O(g) \\ 0 < c \leq \infty &\Rightarrow f = \Omega(g) \end{aligned}$$

- **Mastertheorem (formal):** Es sei $T(1) = c$, $T(n) = a \cdot T(\frac{n}{b}) + f(n)$ mit $n = b^{x \in \mathbb{Z}}$, $b \in \mathbb{N}_{>1}$, $a \geq 1$, $c > 0$ und $f : \mathbb{N} \to \mathbb{R}$, $\epsilon > 0$:

$$\begin{aligned} f(n) = O(n^{(\log_b a)-\epsilon}) &\Rightarrow T(n) = \Theta(n^{\log_b a}) \\ f(n) = \Theta(n^{\log_b a}) &\Rightarrow T(n) = \Theta(n^{\log_b a} \cdot \log_b n) \\ f(n) = \Omega(n^{(\log_b a)+\epsilon}), a \cdot f(\frac{n}{b}) \leq \alpha f(n), lpha > 1 &\Rightarrow T(n) = \Theta(f(n)) \end{aligned}$$

1.2 Elementare Datenstrukturen

- **Bäume allgemein:** „Ein wesentliches Hilfsmittel zur Strukurierung von Daten". Ein Baum $T = (V, E)$ mit $E \subseteq V \times V - \{(i,i)|i \in V\}$ besitzt einen Knoten ohne Vater (**Wurzel**). Jeder Knoten hat genau einen Vater und maximal k Kinder, wenn es sich um einen k-ären Baum handelt. Die **Tiefe** eines Baumes ist die Länge des längsten Weges von von r nach v (also die maximale Anzahl an Kanten auf dem Weg). Die **Höhe** eines Baumes ist die Länge des längsten Weges von von v zu einem Blatt. T heißt geordnet, wenn das $x-te$ Kind seines Vaters zu erkennen ist.

  Operationen: Wurzel(), Vater(v), Kinder(v), Tiefe(V), Höhe(v), Baum($v, T_1, \ldots, T_m$), Suche(x)

  Implementierungen: Array (Vaterrepräsentation), Adjazenzliste (Zeiger auf (sortierte) Kinder - gut bei Graphen!), Binärbaum (Zeiger auf linkes bzw. rechtes Kind - gut bei Binären Bäumen), Kind-Geschwister (Zeiger auf erstes (linkestes) Kind und auf nächsten Geschwister-Knoten - gut bei allgemeinen Bäumen)

  Durchlaufstrategien: Postorder, Preorder, Inorder (rekursiv oder mit Stack) in Zeit $O(n)$

- **Topologisches Sortieren:** „Datenstruktur ähnlich der Queue, nur möchte man nicht das älteste, sondern das Element mit der höchsten Priorität entfernen"

  Operationen: insert(Priorität), delete_max(), change_priority(wo, neue Priorität), remove(wo)

- **Heaps: Struktur:** Bei einem Baum T der Tiefe t haben alle Knoten der Tiefe höchstens $t-2$ genau 2 Kinder. Wenn ein Knoten v der Tiefe $t-1$ weniger als 2 Kinder hat, so haben alle Knoten rechts von v kein Kind. Wenn v genau ein Kind hat, so handelt es sich dabei um ein linkes Kind.

  Heap-Ordnung: Baum T (geordnet und binär), wobei jeder Knoten eine Priorität $p(v)$ besitzt und für jedes Kind w von v gilt: $p(v) \geq p(w)$

  Implementierung: Array, bei dem $H[1] = p(r)$ die Wurzel ist und wenn $H[i]$ die Priorität von v speichert, dann speichert $H[2 \cdot i] = p(v_L)$ und $H[2 \cdot i + 1] = p(v_R)$. Beim Einfügen von Knoten muß ein **repair_up** (nach oben schieben eines Knoten, bis der Vater größer ist als der Knoten) und beim Entfernen ein **repair_down** (Vertauschen der Wurzel mit dem letzten Kind, dann nach unten sinken lassen, indem man immer mit dem größten Kind vertauscht) durchgeführt werden.

Aufwand: insert, delete_max, change_priority, remove in $O(\log_2 n)$, Build-heap in $O(n)$ und Heapsort in $O(n \log_2 n)$

Anwendung: Im Disjkstra-Algorithmus

1.3 Sortieren

- **Bubblesort:** Erwartete Laufzeit / best- und worst-case $\Theta(n^2)$, da alle Permutationen gleichwahrscheinlich. Gut für fast sortierte Eingaben (Elemente, die bereits in der Nähe Ihrer endgültigen Position stehen). Auch gut bei wenig Zahlen.

- **Selectionsort:** Laufzeit wie Bubblesort - gut bei kurzen Arrays mit langen Datensätzen

- **Insertionsort:** Laufzeit wie Bubblesort - gut bei kleinen Datenmengen und fast sortierte Eingabefolgen

- **Quicksort:** Wichtig ist die Funktion `partition(p, links, rechts)` und vor allem die Pivot-Wahl-Strategie.
  Die Invariante bei `partition` ist zu Anfang *„Alle Zellen links von l (einschließlich l) besitzen nur Zahlen $\leq \alpha$, alle Zellen rechts von r (einschließlich r) besitzen nur Zahlen $\geq \alpha$."* Die Laufzeit ist im worst-case $\Theta(n \log_2 n)$ und im best-case $\Theta(n \log_2 n)$
  Verbessert werden kann die Laufzeit durch die Wahlstrategie für das Pivot-Element z.B. durch Wahl des Median aus $A[1]$, $A[\frac{n}{2}]$ und $A[n]$ oder durch Zufallswahl (in diesem Fall ist die erwartete Laufzeit $\Theta(n \log n)$ für jede Eingabe). Die Zufallszahl kann z.B. mittles linearer Kongruenzen gewählt werden: $x_{i+1} = (a \cdot x_i + b) \mod m$, wobei m prim sein sollte.
  Weitere Verbesserungen kann bringen: Beseitigung der Rekursion und Herunterdrücken der Rekursionstiefe von n auf $n \log n$ (Implementierung mittels Stack, wobei immer das längere Teilproblem auf den Stack gelegt wird $\rightarrow$ die Stackhöhe kann $\log n$ nicht überschreiten) und Sonderbehandlung kleiner Teilarrays (Teilprobleme ≤ 25 werden mittels Insertion-Sort sortiert).
  Auch das Auswahlproblem kann so einfach implementiert werden, so daß es in $O(n)$ gelöst werden kann.
  Offene Fragen:

  - Ist QS ein $in-place$-Algorithmus?
  - Gibt es einen Zusammenhang zwischen QS und Binärbaumsuche?
  - Mergesort hat bessere Konstante gegnüber QS, dafür aber doppelter Speicherplatz!

- **Mergesort:** Besonders geeignet für **externes Sortieren** (Daten auf Platte oder Band). Die Eingabefolge wird in zwei gleich große Folgen zerlegt, die dann sortiert und wieder **zusammengemischt** werden.
  Die best-, worse- und average-case Laufzeiten sind alle gleich $\Theta(n \log n)$, da die Rekursionsgleichung $M(n) = 2M(\frac{n}{2}) + c \cdot n$ ist. Der Rekursionsbauom wird in Postorderreihenfolge abgearbeitet, da zuerst die Teilprobleme gelöst werden und dann die Lösungen zusammengesetzt werden. Ein großer Nachteil im Vergleich zu Quicksort ist die Verwendung eines zusätzlichen Arrays zum Mischen der Daten.

- **Ausgeglichenes Mehrwegemischen:** Es stehen $2b$ Bänder zur Verfügung, wobei m der n Daten simultan im Hauptspeicher gehalten werden können. Beim *Vorsortieren* werden alle n Daten in Blöcken der Länge n/m sortiert und auf die Bänder $i \mod b$ zurückgeschrieben. Dann werden die Daten durch *Minimumbestimmung* sortiert auf Band b sortiert gespeichert, und zwar ein Block nach dem anderen. Zuletzt enthalten die Bänder $b, b+1, \ldots, 2b-1$ sortierte Blöcke der Länge m, die dann in er nächsten Phase zu den neuen Eingabebändern werden.
  Laufzeit: Bei n Daten, $2b$ Bändern und Hauptspeicherkapazität m benötigt das Vorsortieren $\lceil \log_b(n/m) \rceil$ Phasen, es wird also auf jede Zahl nie mehr als $1 + \lceil \log_b(n/m) \rceil$-mal zugegriffen.

- **Laufzeit Vergleichs-orientierter Sortierverfahren - Vergleichsbaum:** Für n Eingaben $x_1 \dots x_n$ ist ein binärer Baum, so daß jedem inneren Knoten ein Vergleich der Form $x_i < x_j$ zugeordnet ist. Für eine Eingabe $(a_1 \dots a_n)$ wird der Baum von der Wurzel an durchlaufen. Der Vergleichsbaum heißt sortiert, wenn alle Eingaben, die dasselbe Blatt erreichen denselben Ordnungstyp besitzen → Sortierbaum. Ein *Ordnungtyp* ist die Permutation π, die die Eingabe a sortiert.
  Da ein Sortierbaum mindestens $n!$ viele Blätter besitzt, hat er also mindestens eine Tiefe von $\log_2(n!) \geq \frac{n}{2} \log_2(\frac{n}{2})$

- **Distribution Counting:** Die zu sortierenden Zahlen dürfen nur aus dem Intervall $[0, m-1]$ sein. in dem Array *Zaehle* wird für jede Zahl die Anzahl derer Vorkommen festgehalten, sodaß die Lauftzeit mit $= (n+m)$ sehr günstig ausfällt.
  DC ist nur sinnvoll für $m = O(n \log_2 n)$.

- **Radix-Exchange:** Die Zahlen werden in b-ärer Darstellung vorgehalten. Damit steigt der Programmieraufwand.
  Zuerst werden die Zahlen in die b-äre Darstellung umgeformt. Dann findet eine *Verteilungsphase* statt, in der die Zahlen auf b Queues verteilt werden (entsprechend der i-ten Ziffer). In der *Sammelphase* werden dann die Elemente der Queues wieder in A entleert.
  Wichtig ist die Stabilität, damit bei gleichen hochwertigen Ziffern die restlichen Ziffern über die Sortierung entscheiden. Die Laufzeit beträgt $O((n+b) \cdot \log_b m)$, wobei man immer $b \leq n$ wählen sollte. Radixsort versagt also für zu große Zahlen.

- **Fragen:**
  - Radixsort
    - (Zahlen von $1 \dots n^k$) - welche Basis? ($b = n$, da $\log_b n^k = k$)
    - Laufzeit?
    - Zahlen aus beschränktem Zahlenintervall
    - Beweis: LZ $O((n+b) \log_b m)$
  - Welche Algorithmen für Zahlen $1 \dots \sqrt{n}$ (→Distribution Counting) und $1 \dots n^7$ (→Radixsort)?
  - Insertion-Sort (Binäres einfügen; Warum k Vergleiche?)
  - Quicksort
    - Divide et impera
    - Implementierung von Quicksort nicht rekursiv (- mit Stack; Was ist mit Höhe des Stacks?)
    - Wie kann man die rek. Aufrufe beim QS optimieren? Gaußsche Glockenkurve
    - Ist QS ein $in-place$-Algorithmus?
    - Erwartete Laufzeit?
    - Gibt es einen Zusammenhang zwischen QS und Binärbaumsuche?
    - Mergesort hat bessere Konstante gegnüber QS, dafür aber doppelter Speicherplatz!
  - Untere Schranke für Sortierung mit Vergleichen (Beweis)?
  - Externes Sortieren?
  - Nicht vergleichsorientierte Algorithmen?
  - Laufzeitanalyse von Mergesort
  - Batcher-Sort (LZ)

1.4 Wörterbuchproblem

Bei einer gegebenen Menge S und einem Schlüssel x möchten wir folgende Mengenoperationen unterstützen:

insert(x): $S = S \cup \{x\}$

remove(x): $S = S - \{x\}$

lookup(x): finde heraus, ob $x \in S$ und greife gegebenenfalls auf den Datensatz von x zu.

select(x): bestimme den k-kleinsten Schlüssel

rang(x): bestimme den Rang von x, wobei rang(x)=k genau dann, wenn x der k-kleinste Schlüssel ist.

interval(a, b): bestimme, in aufsteigender Reihenfolge, alle Schlüssel $y \in S$ mit $a \leq y \leq b$.

Ein **statisches Wörterbuch** besteht nur aus der Operation *lookup* und kann z.B. mittels eines sortierten Arrays oder einer sortierten Liste implementiert werden (lookup benötigt dann $O(n \log_2 n)$ Zeit).
Ein **(nicht geordnetes) Wörterbuch** besteht dabei nur aus den Operationen *insert, remove, lookup.* Als Datenstrukuren können folgende verwendet werden:

- **Binäre Suchbäume: Definition:** Sei T ein geordneter binärer Baum, so daß jedem Knoten v von T ein Paar $daten(v) = (Schlüssel(v), Info(v))$ zugeordnet sei. Für jeden Schlüsselwert x in T gibt es dann höchstens einen Knoten v mit $Schlüssel(v) = x$ und für jeden Knoten v_{links} im linken Teilbaum von v und jeden Knoten v_{rechts} im rechten Teilbaum von v ist $Schlüssel(v_{links}) < Schlüssel(v) < Schlüssel(v_{rechts})$.

  Laufzeit: Die erwartete Laufzeit ist logarithmisch (da jede Permutation der Schlüssel gleichwahrscheinlich ist, und somit gilt $L(n) = \frac{1}{n}\sum_{x \in S} T(x) \Rightarrow L^*(n) = n + \frac{1}{n}\sum_{k=1}^{n}(L^*(k-1) + L^*(n-k))$). Das erst eingefügte Element spielt also eine ähnliche Rolle wie das Pivotelement im Quicksort. Wir erhalten eine best-case Laufzeit für tiefenbalancierte Bäume, in denen sich die Tiefe um höchstens 1 unterscheidet.
  Damit ist die worst-case Laufzeit von insert, remove und lookup $\Theta(Tiefe(T))$. Wenn T n Knoten hat, dann ist bestenfalls Tiefe(T)=$\lfloor \log_2 n \rfloor$ und schlechtestenfalls Tiefe(T)=$n-1$.
  Die erwartete Laufzeit einer erfolgreichen lookup-Operation in binären Suchbäumen von n Schlüsseln ist $O(\log_2 n)$.

- **AVL-Bäume: Definition:** Sei T binärer Suchbaum, dann ist T AVL-Baum, wenn für jeden Knoten v mit linkem Teilbaum $T_L(v)$ und rechtem Teilbaum $T_R(v)$ gilt: $|Tiefe(T_L(v)) - Tiefe(T_R(v))| \leq 1|$. Dabei heißt $b(v) = Tiefe(T_L(v)) - Tiefe(T_R(v))$ der **Balance-Grad** von v (im AVL-Baum stets $b(v) \in \{-1, 0, 1\}$)
  Bei n Knoten hat ein AVL-Baum höchstens die Tiefe $2\log_2 n$.

  Operationen: Anstatt gleich ein *remove* durchzuführen, kann ein *lazy-remove* durchgeführt werden, bei dem die gleöschten Knoten lediglich markiert werden und erst gelöscht werden, wenn mehr als 50 als gelöscht markiert sind.
  Bei einem *insert* wird der Schlüssel wie beim binären Suchbaum nach Suche einfach eingefügt. Ist dann die Balance-Eigenschaft verletzt, so führen wir Rechts- bzw. Linksrotationen aus.

  Laufzeit: Die Operationen *lookup* und *insert* haben die worst-case Laufzeit $O(\log_2 n)$.

- **Splay-Bäume: Vorteile:** Benötigen keine zusätzrlichen Informationen, geringer Programmieraufwand, „passen sich den Daten an", amortisierte Laufzeit $O(n \log_2 n)$

  Nachteil: miserable worst-case Laufzeit $\Theta(n)$,

  Operationen: Die Splay-Operatione führt zuerst eine binäre Suche nach x durch, bringt y, mit entweder $y = x$ oder y größter Schlüssel kleiner x, an die Wurzel. Alle Operationen lassen sich auf die Splay-Operation zurückführen.

Laufzeit: Wiederholte Anwendung der Splay-Operation führt dazu, daß die Knoten des Suchpfades der Wurzel fast um die Hälfte näher rücken ($\rightarrow$ **Suchpfadeigenschaft**).
Wenn n Splay-Operationen in einem anfänglich leeren Splay-Baum durchgeführt werden, so beträgt die worst-case Laufzeit $O(n \log_2 n)$.

- (a,b)-**Bäume: Definition:** $a, b \in \mathbb{N}$ mit $a \geq 2$ und $b \geq 2a-1$. Baum T hat dann (a,b)-Eigenschaft, wenn alle Blätter von T die gleiche Tiefe besitzen, alle Knoten höchstens b und mindestens a Kinder haben (nur die Wurzel hat mindestens zwei Kinder).
Ein (a,b)-Baum hat (a,b)-Eigenschaft, wobei jeder Schlüssel S in genau einem Knoten gespeichert wird und jeder Knoten seine Schlüssel in aufsteigender Reihenfolge speichert. Jeder innere Knoten speichert dabei genau $k-1$ Schlüssel, und jedes Blatt zwischen $a-1$ und $b-1$). Ein innerer Knoten v mit Schlüsseln $x_1, \ldots, x_c$ speichert dann in dem i-ten Teilbaum $(2 \leq i \leq c)$ Schlüssel aus dem Intervall (x_{i-1}, x_i), bzw. $(-\infty, x_1)$ und (x_c, ∞).

  B-Bäume: (a,b)-Bäume mit $b = 2a-1$ heißen B-Bäume. Sie sind besonders gut, wenn die Daten auf einem Hintergrund-Speicher abgelegt sind, da höchstens $O(\log_a n)$ Knoten bei der Ausführung einer Wörterbuch-Operation besucht werden.

  Operationen: Bei einem **insert** kann es vorkommen, daß ein Knoten zuviele Schlüssel bekommt. In diesem Fall wird ein neuer Knoten v mit v_{links} Knoten $x_1, \ldots, x_{\lceil b/2 \rceil - 1}$ und v_{rechts} Knoten $x_{\lceil b/2 \rceil + 1}, \ldots, x_b$ („bottom-up").
Beim „top-down" zerlegen wir alle Knoten auf dem Weg mit b Kindern, was zu unnötigen Tiefenerhöhungen führen kann.
Bei **remove** kann es passieren, daß ein Knoten zu wenig Schlüssel hat. Dann versuchen wir von dem linken oder rechten Geschwisterknoten (mit mindestens a Schlüsseln) einen zu klauen. Wenn dies nicht geht, weil beider Geschwisterknoten nur $a-1$ Schlüssel haben, so veschmelzen wir den Knoten mit einem Geschwisterknoten, so daß ein neuer Knoten mit maximal $(a-1)+1+(a-2) = 2a-2 \leq b-1$ Schlüsseln entsteht. Wenn jetzt ein Schlüssel zu wenig im Knoten bleibt, müssen wir die Prozedur rekursiv nach oben fortsetzen.

  Laufzeit: Für einen (a,b)-Baum gilt $\log_b(n) - 1 < Tiefe(t) < \log_a(\frac{n-1}{2}) + 1$
Für die *lookup*- und *insert* Operation benötigen wir damit $Tiefe(T)+1$ und für die *remove*-Operation $4(Tiefe(T)+1)$ Speicherzugriffe.

- **Hashing:** Wenn die Menge U von vorneherein bekannt ist und in den Hauptspeicher paßt, so ist das Wörterbuchproblem trivial und kann durch eine **Bitvektor-Datenstruktur** gelöst werden (Array , daß für jedes $u \in U$ speichert, ob u präsent ist). Beim Hashing gibt es eine Funktion $h : U \rightarrow \{0, 1, \ldots, m-1\}$, wobei m die Größe einer (internen) Tabelle ist. Da meistens gilt $m < |U|$, kann es zu Kollisionen kommen, d.h. $h(u) = h(v) \forall u, v \in U$. Hierauf kann verschieden reagiert werden:

  Hashing mit Verkettung: Jede Zelle i enthält einen Zeiger auf eine Liste mit Elementen (anfänglich leer).
Bei einer Hashfunktion $h(x) = x \mod m$ haben sich Primzahlen für m als sehr gute Wahl erwiesen. Dabei ist $\lambda = \frac{n}{m}$ der **Auslastungsfaktor** der Tabelle. Die worst-case-Laufzeit ist $\Theta(n)$, während die erwartete Laufzeit höchstens $2 \cdot (1+\lambda)$.

  c-universell: Eine Menge $H \subseteq \{h | h : U \rightarrow \{0, \ldots, m-1\}\}$ ist c-universell, falls $\forall x, y \in U$ mit $x \neq y$ gilt: $|\{h \in H | h(x) = h(y)\}| \leq c \cdot \frac{|H|}{m}$. Wenn H c-universell ist, dann gibt es keine zwei Schlüssel, die mit Wahrscheinlichkeit größer als $\frac{c}{m}$ auf die gleiche Zelle abgebildet werden.
Beispiel: $U = \{0, 1, 2, \ldots, p-1\}$ mit p prim und $H = \{h_{a,b} | 0 \leq a, b < p, h_{a,b}(x) = ((ax+b) \mod p) \mod m\}$ mit $c = (\lceil \frac{p}{m} \rceil / \frac{p}{m})^2$

  Universelles Hashing: Zufälliges Auswählen einer Hashfunktion $h \in H$. Damit wird verhindert, daß man für die Hashfunktion eine worst-case Laufzeit von $\Theta(n)$ erzwingen kann. Durch Wahle einer c-universellen Klassen von Hashfunktionen ergibt sich dann eine erwartete Laufzeit für n Operationen von höchstens $n(1 + \frac{c}{2} \cdot \frac{n}{m})$.

Hashing mit offener Adressierung: Wenn $h(x) = i$ besetzt ist, wird mit einer weiteren Hash-Funktion aus $h_0, \ldots, h_{m-1} : U \to \{0, \ldots, m-1\}$ gehasht. Dieses Verfahren ist speichereffizienter als mit Verkettung, allerdings muß vorher eine gute obere Schranke der maximalen Anzahl einzufügender Elemente bekannt sein (man kann aber auch z.B. die Tabellen sukzessive um den Faktor 2 wachsen lassen). Bei fast gefüllten Tabellen ist die Laufzeit allerdings sehr schlecht.

Beispiel: $h_i(x) = (x + i) \mod m$ („lineares Austesten") oder $h_i(x) = (f(x) + i \cdot g(x)) \mod m$ mit f und g Hashfunktion und m am besten prim (da $\{h_i(x) | 0 \leq i < m\} = \{0, \ldots, m-1\}$), z.B. mit $f(x) = x \mod m$ und $g(x) = m^* - (x \mod m^*)$ mit $m^* < m$.
Problem: Was ist, wenn bei lookup im ersten Schritt der Schlüssel nicht gefunden wird? -Man sollte also diese Hashing- Strategie nicht verwenden, wenn wahrscheinlich viele Schlüssel wieder entfernt werden. Die erwartete Laufzeit einer erfolglosen lookup-Operation ist dann durch $\frac{1}{1-\lambda}$ nach oben beschränkt.

- Fragen:

- Hashing:
  - Was ist Hashing? (1000 Kunden - wie hashen?)
  - Double Hashing? (Funktion hinschreiben) Wie sieht Hashfkt. $h(x) = x \mod m$ aus?
  - Hashing mit Verkettung? (detailliert; Belegungsdichte=erwartete Tabellenlänge)
  - Laufzeit für Chaining?
  - Erwartete Laufzeit für Hashing mit Chaining?
  - Hashen mit offener Adessierung (mögliche Kollisionsbeseitigung; Laufzeit; gute Hashfunktionen?)
  - Wann Hashing statt AVL-Bäume?
  - Direktes Hashing? Bitvektor?
  - Was ist λ? ($\lambda > 1$ nur für verkettete Listen)
  - Laufzeit für Hashingsort $O(1 + \alpha)$, da zunächst geprüft werden muß, ob das einzufügende Objekt bereits enthalten ist!
  - Definition: c-universell (Beispiel)
  - Vergrößerung der Hash-Tabelle
- Bäume allgemein
- Für was sind B-Bäume gut?

1.5 Graphalgorithmen

- Ungerichteter Graph: $G = (V, E)$ besteht aus einer endlichen Knotenmenge V und einer Kantenmenge $E \in \{\{i, j\} | i, j \in V, i \neq j\}$

- Gerichteter Graph: $G = (V, E)$ besteht aus einer endlichen Knotenmenge V und einer Kantenmenge $E \in \{(i, j) | i, j \in V, i \neq j\}$

Adjazente Knoten: Knoten, die durch eine Kante verbunden sind

Inzidenz: Eine Kante ist mit einem Knoten inzident, wenn der Knoten ein Endpunkt der Kante ist.

Weg in G**:** Eine Folge von Knoten $(v_0, v_1, \ldots, v_m)$, für die für jedes i $(0 \leq i \leq m)$ gilt, daß $(v_i, v_{i+1} \in E$ (bei gerichteten Graphen) und $\{v_i, v_{i+1}\} \in E$ (für ungerichtete Graphen.

Weglänge: ist die Anzahl der Kanten wobei ein Weg **einfach** heißt, wenn kein Knoten zweimal auftritt und **Kreis**, wenn der Endknoten ungleich dem Anfangsknoten und der Weg einfach ist.

Azyklisch: ist ein gerichteter Graph, wenn er keine Kreise besitzt.

Zusammenhängend: heißt für einen ungerichteten Graph, daß es für je zwei Knoten $v, w \in V$ einen Weg von u nach v gibt.

Stark zusammenhängend: ist ein gerichteter Graph, wenn es für je zwei Knoten $v, w \in V$ sowohl einen Weg von v nach w als auch von w nach v gibt.

Induzierter Graph: Sei G ein ungerichteter Graph, dann gilt für $V' \subseteq V$, daß $G(V')$ der von V' induzierte Graph ist. D.h., daß $G(V')$ die Knotenmenge V' und alle Kanten von G besitzt, deren Endpunkte in V' liegen.
Wenn $G(V')$ zusammenhängend ist und V' eine größte zusammenhängende Knotenmenge bzgl. der Inklusion ist (wenn also die Hinzunahme jedes weiteren Knotens den Zusammenhang stört), nennen wir V' eine **Zusammenhangskomponente** von G.

Knoten-Beziehungen: Für ungerichtete Graphen gilt: $\{u, v\} \in E \Rightarrow u$ ist **Nachbar** von v. **Grad(v)** beschreibt die Anzahl der Nachbarn von v.
Bei gerichtete Graphen ist: u der **Vorgänger** von v, wenn $(u, v \in E$, und die Anzahl der Vorgänger **outgrad(v)**

- Implementierung von Graphen: Mittels einer **Adjazenzmatrix** A_G mit

$$A_G[u, v] = \begin{cases} 1 & \text{, wenn } \{u, v\} \in E \\ 0 & \text{, sonst} \end{cases}$$

mit Platzbedarf $\Theta(n^2)$ und Zeit für Bestimmung der Nachbarn bzw. Vorgängern $O(n)$.
Bei der **Adjazenzliste** listet das element $A[v]$ alle Nachbarn bzw. Nachfolger von v auf mit Speicherbedarf $\Theta(n + |E|)$. Diese Darstellung ist vor allem bei dünnen Graphen der Adjazenzmatrix überlegen und liefert eine schnellere Implementierung der Nachbar- oder Nachfolger-Operation.

- Königsberger Brückenproblem / Euler-Kreis: Finde einen Weg in einem Graphen, der jede Kante genau einmal durchläuft und an seinem Anfangspunkt endet.

- Tiefensuche (DFS): Ist ein Preorderlauf (+ Markierung) auf (un-)gerichteten Graphen. Um zu verhindern, daß Knoten zweimal besucht werden, wir ein boolesches Array benötigt, in dem vermerkt wird, ob ein Knoten bereits besucht wurde. Die Laufzeit beträgt $O(|V|+|E|)$. Damit kann in $O(|V| + |E|)$ überprüft werden, ob G zusammenhängend ($\Leftrightarrow$ $tsuche(0)$ besucht alle Knoten) und ein Baum ist ($\Leftrightarrow$ es existieren keine Rückwärtskanten).
Wenn v ein Knoten in G ist, dann enthält der Baum von v alle Knoten der Zusammenhangskomponente von v.
Zur Implementierung können wir, sofern der Algorithmus nicht rekursiv laufen soll, einen Stack verwenden.

Kantenbezeichnungen: **Ungerichtete Graphen** Kanten, die im Aufrufbaum der Tiefensuche, und im Graphen enthalten sind heißen **Baumkanten**. **Rückwärtskanten** würden im Graphen einen Knoten mit einem seiner Vorgänger verbinden
Gerichtete Graphen Baumkanten, Rückwärtskantenm, Vorwärtskanten (würden im Aufrufbaum einen Knoten mit einem Nachfolger verbinden) und Querkanten (verbinden zwei Knoten im Aufrufbaum, die weder Vor- noch Nachfahre füreinander sind.

Eigenschaften der Tiefensuche: **Ungerichtete Graphen** Sei $G = (V, E)$, $(u, v) \in E$ und W_G der durch die Tiefensuche auf G definierte Wald der Aufrufbäume. Wenn $tsuche(u)$ vor $tsuche(v)$ aufgerufen wird, dann is v ein Nachfahre von u (in W_G). u und v gehören immer zum selben Baum in W_G und G besitzt nur Baum- und Rückwärtskanten. **Gerichtete Graphen** G azyklisch $\Leftrightarrow$ G besitzt keine Rückwärtskanten und $e = (u, v)$ is eine Baumkante $\Leftrightarrow$ $tsuche(v)$ wird unmittelbar aus $tsuche(u)$ aufgerufen. Um die Kanten sicher erkennen zu können muß dem Algorithmus ein Integer-Array $Anfang\_nr$ und $Ende\_nr$ zur Verfügung gestellt werden (anstelle von $besucht$), in denen am Anfang und am Ende der Rekursion die Knotennummern abgespeichert werden.

- **Breitensuche (BFS):** Für jeden Knoten werden zuerst alle Nachfolger besucht, gefolgt von der Generation der Enkel usw. Die Knoten werden also in der Reihenfolge ihres Abstandes von v, dem Startknoten der Breitensuche, erreicht.
Zur Implementierung verwenden wir eine Queue, in der wir die unbesucheten Nachbarknoten eintragen und markieren. Die Laufzeit beträgt höchstens $O(|V_w| + |E_w|)$, womit BFS genauso schnell ist wie DFS, definiert aber den Baum T_w von kürzesten Wegen für G.

- **SSSP-Problem (Disjkstra):** Jeder Kante eines Graphen wird durch die Funktion *länge* : $E \to \mathbb{R}_{\geq 0}$ eine Länge ≥ 0 zugewiesen.
Sei $S \subseteq V$ und $s \in S$, dann nehmen wir an, daß wir für jeden Knoten $w \in V - S$ den Wert **distanz(w)**= Länge eines kürzesten Weges W von s nach w, so daß W, mit Ausnahme des Endknotens w, nur Knoten in S durchläuft kennen. Der Algorithmus sucht nun einen Knoten $v \in V - S$ mit kleinstem Distanzwert und behauptet, daß distanz[v]=Länge eines kürzesten Weges von s nach v ist. Der Knoten wird dann der Menge S hinzugefügt und die Distanz-Werte für Knoten in $V - S$ werden aktualisiert. Knoten, die von S aus gar nicht erreichbar sind haben einen Distanzwert von ∞.
Algorithmus:

  - Setze $S = \{s\}$ und

$$distanz[v] = \begin{cases} \text{länge}(s, v) & \text{wenn } (s, v) \in E \\ \infty & \text{sonst.} \end{cases}$$

  - Solange $S \neq V$ wiederhole
    - wähle einen Knoten $w \in V - S$ mit kleinstem Distanzwert
    - Füge w in S ein.
    - Berechne die neuen Distanz-Werte der Nachfolger von w. Insbesondere setze für jeden Nachfolger $u \in V\ S$ von w

$$c = distanz[w] + laenge(w, u);$$
$$distanz[u] = (distanz[u] > c)\ ?\ c : distanz[u];$$

  Die Korrektheit kann man mit Hilfe eines konstruierten Weges beweisen (erster Knoten, der nicht mehr zu S gehört, weiter weg als w usw.). Die Laufzeit beträgt $O((|V| + |E|) \log_2 |V|)$ (da $|V| - 1$-mal Knoten mit kleinstem Distanz-Wert gewählt und die neuen Distanzwerte der Nachfolger berechnet werden; der kleinste Distanz-Wert wird durch Verwendung eines Heaps berechnet, wobei die Priorität durch den Distanz-Wert definiert ist und die Ordnung umgekehrt wird, damit die Operationen `insert` und `deletemin` unterstützt werden; es werden maximal $|E|$ Neuberechnungen durchgeführt und der Heap besitzt nie mehr als $|V| + |E|$ Elemente, d.h. eine Heap-Operation benötigt Zeit $O(\log_2(|V| + |E|)) = O(\log_2 |V|)$).
  Der Dijkstra Algorithmus ersetzt also im wesentlichen nur die Schlange der Breitensuche durch eine Prioritätswarteschlange. Die kürzesten Wege selbst können über eine Speicherung des unmittelbaren Vorgängers rekonstruiert werden.

- **Minimale Spannbäume:** Sei $G = (V, E)$ ein ungerichteter Graph. Ein Baum $T = (V', E')$ heißt Spannbaum, falls $V' = V$ und $E' \subset E$.

  Prim's Algorithmus: Mit Laufzeit $O(|E| \cdot \log_2 |V|)$. Dabei werden die kreuzenden Kanten in einem Heap verwaltet und über die `deletemin`-Operation entfernt.

  - Setze $S = \{0\}$ und $E' = \emptyset$
  - Solange $S \neq V$, wiederhole:
    Bestimme eine kürzeste kreuzende Kante $e \in E$, die also einen Endpunkt in S und einen in $S - V$ besitzt und eine länge(e)=min{länge(e')—$e' \in E$ und e' besitzt jeweils einen Endpunkt in S und $V - S$}
    Sei $E = \{u, v\}$ mir $u \in S$ und $e \in V - S$. Setze $S = S \cup \{v\}$ und $E' = E' \cup \{e\}$.

Kruskals Algorithmus: Mit Laufzeit $O(|E| \cdot \log_2 |V|)$. Der Wald wird dabei mittels eines Arrays und der Vater-Repäsentation gespeichert. Die Operation `union(i,j)` hängt dann immer den kleineren Baum unter die Wurzel des größeren (damit kann der Baum höchstens $\log_2(|V|)$ tief werden).

- Sortiere die Kanten gemäß aufsteigendem Längenwert. Sei $W = (V, E')$ der leere Wald, also $E' = \emptyset$.
- Solange W kein Spannbaum ist, wiederhole:
  Nimm die gegenwärtig kürzeste Kante e und entferne sie aus der sortierten Folge und verwerfe e, falls e einen Kreis in W schließt. Anonsten akzeptiere e und setze $E' = E' \cup \{e\}$

- Fragen:
- Welcher Alg. im ungewichteten Graph für "Kürzeste Wege" und "Bipartiter Graph"?
- Warum funktionieren Dijkstra und Prim
- Dijkstra-Algorithmus für kürzeste Wege?
  - Datenstruktur?
  - Definition von $distanz[v]$?
  - Korrektheitsbedingung?
  - Wie Distanzen im Heap aktualisieren?
  - Wo sind Informationen enthalten?
  - Mit Prioritätswarteschlange vgl. mit BFS?
  - Was ist wenn alle $distanz = 1$ haben? (BFS)
  - Wie implementiert man V und $V \backslash S$?
  - Was wissen wir über $w \in V \backslash S$?
- Kruskal-Algorithmus (minimale Spannbäume)
  - detailliert - wie funktioniert $union()$? (muß schnell sein!)
  - wie wähle man m? -Prim!
  - Datenstruktur für Kruskal? (Vaterrepräsentation)
  - Funktioniert der Algorithmus auch mit negativen Kanten? -Ja (aber nicht Dijkstra)
- Algorithmus von Prim?
- Wie stelle ich Bipartiten Graph fest?
- Ford-Fulkerson (Minimaler Schnitt = Maximaler Fluß)
- Kantentypen bei DFS
- Fouriertransformation, n-te Einheitswurzel

1.6 Entwurfsmethoden für Algorithmen

- Divide and Conquer: Das gegebene Problem wird in kleinere Teilprobleme zerlegt, deren Lösung eine Lösung des Ausgangsproblems bedingt (Beispiel: Binärsuche, Quicksort/Mergesort, Durchsuchungsmethoden für Bäume und Graphen)

Multiplikation ganzer Zahlen: Es werden nur drei Multiplikationen benötigt, was zu einer Laufzeit von $O(n^{\log_2 3})$ führt)

Multiplikation von Matrizen: Zerlegung der Matrizen in vier gleich große Teilmatrizen und Durchführung von 7 Matrizenmultiplikationen, Laufzeit $\Theta(n^{\log_2 7})$, erst sinnvoll ab mehreren hundert Zeilen/Spalten

- Dynamisches Programmieren: Im Unterschied zu Divide and Conquer werden Lösungen für Teilprobleme mehrmals benötigt, deshalb werden sie abgespeichert.

Floyd's Algorithmus zur Bestimmung aller kürzesten Wege: Es muß das Teilproblem

$$distanz_k[u][v] = \text{Länge des kürzesten Weges von u nach v,}$$

dessen innere Knoten nur aus Knoten in $\{0, 1, \ldots, k\}$ bestehen berechnet weden. Dabei ist das einfachste Problem

$$distanz_{-1}[u][v] = \begin{cases} \text{länge}(u,v) & (u,v) \in E \\ \infty & \text{sonst} \end{cases}$$

wobei sich damit rekursiv ergibt:

$$distanz_k[u][v] = min\{distanz_{k-1}[u][v], distanz_{k-1}[u][k] + distanz_{k-1}[k][v]\}$$

Zur Implementation verwenden wir ein 3-dimensionales Array distanz., wobei angenommen wird, daß G als Adjazenzmatrix vorliegt. Wegen einer Laufzeit von $\Theta(n^3)$ empfiehlt sich der Algorithmus vor allem für dichte Graphen, da sonst Dijkstra eine bessere Alternative darstellt.

Warshall's Algorithmus für die Berechnung der transitiven Hülle eines gerichteten Graphen: Hier kann man ein Teilproblem als $H\ddot{u}lle_k[u,v] = 1 \Leftrightarrow$ Es gibt einen Weg von u nach v, dessen innere Knoten nur aus Knoten in $\{0, \ldots, k\}$ bestehen. Damit ist

$$H\ddot{u}lle_{-1}[u,v] = \begin{cases} 1 & (u,v) \in E \\ 0 & \text{sonst.} \end{cases}$$

mit dem rekursiven Schritt

$$H\ddot{u}lle_k[u,v] = H\ddot{u}lle_{k-1}[u,v] \vee (H\ddot{u}lle_{k-1}[u,k] \wedge H\ddot{u}lle_{k-1}[k,v])$$

Was dann zu einer Laufzeit von $O(|V|^3)$ führt.
Auch hier ist die n-malige Anwendung der Tiefensuche ein scharfer Konkurrent, weshalb der Algorithmus nur für dichte Graphen ($|E| = \Omega(|V|^2)$ benutzt werden sollte.

Multiplikation von n Matrizen: Da die Klammerung hier eine sehr wichtige Rolle spielt, läßt sich der Rekursionsschritt durch

$$Kosten[i][i+k+1] = min\{Kosten[i,s] + Kosten[s+1, i+k+1] + r_{i-1} \cdot r_s \cdot r_{i+k+1} \mid i \leq s \leq i+k\}$$

definieren. Damit wird Zeit $O(n^3)$ benötigt.

- **Greedy-Algorithmen:** Konstruieren den Lösungsvektor Komponente nach Komponente, wobei ein gesetzter Wert niemals zurückgenommen wird. Beispiele sind Prim, Kruskal und Dijkstra.

  Non-preemptive Scheduling: Mehrere Aufgaben mit Startzeiten s_i und Terminierungszeiten t_i. Implementierung: Alle Aufgaben nach Startzeit sortieren und dann Aufgaben der Reihe nach auswählen und immer die löschen, deren Startzeit vor der gewählten Terminierungszeit liegt $\rightarrow$ Laufzeit $O(n \log_2 n)$.

  Huffman-Codes: Präfixcode, der auch durch einen Binärbaum dargestellt werden kann. Die Implementierung erfolgt mittels eines Heaps , wobei für je zwei (minimale) Buchstaben, die entfernt werden, wieder ein neuer mit der Summe hinzugefügt wird. Die Laufzeit beträgt $O(n \log_2 n)$

  Binpacking-Problem: Da das Problem NP-Vollständig ist, kann ein Greedy-Algorithmus mittels best-fit.

- **Backtracking:** Es wird versucht, systematisch alle Lösungen im Lösungraum systematisch aufzulisten (Ziel: Zusammenfassung äquivalenter Lösungen, frühzeitige Erkennung, daß bestimmte partielle Lösungen nicht zu einer Gesamtlösung erweiterbar sind).

- **Branch and Bound:** Für jede partielle Lösung werden gute untere und obere Schranken benötigt.

2 Fragen zu Theoretischer Informatik 2

2.1 Grundlagen

Alphabet Σ: Menge von Buchstaben. $\Sigma^n = \{(a_1, \ldots, a_n) | a_i \in \Sigma\}$ ist die Menge aller Worte der Länge n (mit $\Sigma^0 = \epsilon$ „leere Wort" und $\Sigma^* = \bigcup_n = 0^\infty \Sigma^n$ „Menge aller Worte über Σ)

Konkatenation: Σ Alphabet, $u, v, \in \Sigma^* \Rightarrow u \cdot v = u_1 \ldots u_n v_1 \ldots v_m = uv$

Teilwort: u Teilwort von $v \Rightarrow \exists u_1, u_2 \in \Sigma^* : v = u_1 \cup u_2$

Präfix: u Präfix von $v \Rightarrow \exists u_1 \in \Sigma^* : v = u1 \cup u_2$

Konkatenation von Sprachen: L_1, L_2 Sprachen $\Rightarrow L_1 \circ L_2 = \{uv | u \in L_1, v \in L_2\}$.
L Sprache über $\Sigma \Rightarrow L^n = \{u_1 \ldots u_n | u_1, \ldots, u_n \in L\}, L^* = \bigcup_{n=0}^{\infty} L^n, L^+ = \bigcup_{n=1}^{\infty} L^n$

Reguläre Ausdrücke: Definiert durch

- $\emptyset, \epsilon$ und $a (\in \Sigma)$ sind regulär (leere Sprache, Sprache des leeren Wortes, Sprache des einbuchstabigen Wortes)
- A_1, A_2 reguläre Ausdrücke $\Rightarrow (A_1) + (A_2), (A_1) \circ (A_2), (A_1)^*$ sind regulär
- Alle regulären Ausdrücke lassen sich durch endlich viele Anwendungen von obigen Regeln erzeugen

Probleme gibt es z.B. mit regulären Klammerausdrücken.

2.2 Berechenbarkeit

Ziel ist die Formalisierung der Berechenbarkeit einer Funktion $F : \Sigma_1^* \to \Sigma_2^*$ bzw. eine Formalisierung der Entscheidbarkeit einer Sprache L.

- **Turingmaschinen:** Formal ein 6-Tupel $(Q, \Sigma, \delta, q_0, \Gamma, F)$.

  Architektur: Beidseitig unendliches Band bestehend aus Speicherzellen mit Adressen aus $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$, die ein Symbol aus dem **Bandalphabet** Γ speichern.

  Eingabekonvention: Eingaben sind Worte über dem **Eingabealphabet** Σ mit $\Sigma \subseteq \Gamma$. Alle Zellen des Bandes, die keine Buchstaben aus dem Eingabealphabet gespeichert haben, speichern das **Blanksymbol** B mit $B \in \Gamma$.

  Programm: Eine Menge von Befehlen $(q, a, q', b, wohin)$ bilden ein Programm mit $q, q' \in Q$, Zustand einer (endlichen) Zustandsmenge Q, $a, b \in \Gamma$ und $wohin \in \{links, bleib, rechts\}$. Ein Programm ist also formal eine Zustandsüberführungsfunktion

  $$\delta : Q \times \Gamma \to Q \times \Gamma \times \{links, bleib, rechts\},$$

  die jedem Paar $(q, a) \in Q \times \Gamma$ nur eine Anweisung zuordnet.

  Ausgabekonvention: Eine Turingmaschine hält auf eine Eingabe $w \Leftrightarrow$ die Maschine hat einen Zustand q erreicht und liest ein Symbol $q \in \Gamma$, so daß $\delta(q, a) = (q, a, bleib)$.
  Funktionsberechnung: Ausgabe ist das Wort bestehend aus allen Buchstaben der gegenwärtig gelesenen Zelle bis (ausschließlich) zu ersten Zelle rechts von der gegenwärtigen Zelle, die B speichert. Formal 5-Tupel $(Q\Sigma, \delta, q_0, \Gamma)$.
  Wenn M eine Funktion berechnet, dann bezeichnet $f_M : \Sigma^* \to \Gamma^*$ mit

  $$f_M(w) = u \Leftrightarrow M \text{ gibt, für Eingaben } w, \text{ die Ausgabe } u$$
  $$f_M(w) = undefiniert \Leftrightarrow M \text{ hält nicht auf Eingabe } w$$

  die von M berechnete Funktion
  Spracherkennung: Menge der akzeptierenden Zustände $F \subseteq Q$. Die von M erkannte Sprache $L(M)$ ist dann $L(M) = \{w \in \Sigma^* | M \text{ akzeptiert } w\}$.

Berechenbarkeit: Eine Funktion $f : \Sigma_1^* \to \Sigma_2^*$ heißt berechenbar $\Leftrightarrow \exists$ TM M $\forall w \in \Sigma_1^*$ hält mit Ausgabe $f(x)$ (also ist $f = f_M$ und $f_M(w) \neq$ undefiniert $\forall w \in \Sigma_1^*$).

k-Band-Turingmaschinen: TM mit k Bändern, wobei die Eingabe auf Band 1 gespeichert ist. Ein Befehl wird in Abhängkeit der Eingabe und den Inhalten der k gelesenen Zellen ausgewählt. Der zu druckende Buchstabe wird für jedes Band separat festgelegt.
Eine k-Band-TM, die auf Eingaben der Länge n nach höchstens $T(N)$ Schritten hält kann durch eine TM M' simuliert werden, die nach höchstens $O(T^2(n))$ Schritten hält, wobei M und M' dieselbe Funktion berechnen bzw. dieselbe Sprache erkennen (Beweis durch Spuren).

TM können das gleiche wie reale Rechner, nur langsamer. Die einzelnen Rechenschritte wirken nur lokal, so daß Rechnungen einfacher zu verfolgen sind. In Fragen der Berechenbarkeit sind TM äquivalent zu allen bekannten Rechnermodellen.

- **Entscheidbarkeit/Rekursivität:** Eine Sprache $L \subseteq \Sigma^*$ entscheidbar $\Leftrightarrow \exists$ TM M, die $\forall x \in \Sigma^*$ hält mit: $x \in L \Leftrightarrow M$ akzeptiert x (also $L = L(M)$ und M hält stets).

- **Registermaschinen:** Besteht aus Eingabe- und Ausgabeband(in Zellen unterteilt), Speicher (Akkumulator und unendlich viele Register speichern ganze Zahlen). Befehle werden durch Label identifiziert (Lese-/Schreib-, Speicher-, arithmetische und Goto-Befehle, Verleichsoperationen). Zu Anfang steht die Eingabe linksbündig auf dem Eingabeband, der Lese-/Schreibkopf steht auf der ersten Zelle und alle Register sind mit 0 initialisiert.
Eine RM kann von einer TM in Zeit $O = (T^6(n))$ Schritten simuliert werden (Beweis durch k-Band TM in Zeit $O(T^3(n))$).
Was RM können, können reale Rechner erst recht. Bei der Rechenzeit müssen allerdings Operationen auf sehr langen Zahlen fair, d.h. mit dem logarithmischen Kostenmaß, bewertet werden. Umgekehrt bieten reale Rechner weit mehr Komfort, wobei sich die auf realen Rechnern möglichen Operationen auf RM ohne wesentlichen Zeitverlust nachbilden lassen. RM sind ein realistischen, assemblernahes Rechnermodell.

- **Churchsche These:** *Die Klasse der intuitiv berechenbaren Funktionen ist gleich der Klasse der Turingberechenbaren Funktionen.* Mit der Existenz von Rechnern ist nachgewiesen, daß die Turing-berechenbaren Funktionen tatsächlich berechenbar sind. Die Umkehrung ist prinzipiell nicht beweisbar, da der Begriff „intuitiv berechenbar" nicht wohldefiniert ist. Nur ein Gegenbsp. könnte die Chruchsche These widerlegen!
$\Rightarrow$ Für Fragen der Berechenbarkeit ist die Wahl des Rechnemodells unerheblich, solange es soviel kann wie Turingmaschinen.

- **Universelle TM:** Eine TM U, die ein (geeignet kodiertes) Turingmaschinenprogramm P und eine Eingabe w für P als Eingabe annimmt und sodann das Programm P auf der Eingabe w simuliert. Dabei kodiere man die Eingabe in das binäre Zahlensystem und speichere die Eingabe in den Zuständen ab. Der Startzustand und gleichzeitig einzige akzeptierende Zustand ist q_0 und die Funktion δ mit ihrer endlichen Funktionstabelle läßt sich durch eine **Gödelnummer** beschreiben:

$$1^{|Q|}0code(\delta)00$$

Die Gödelnummer der TM M wird mit $< M >$ bezeichnet. und beschreibt das Programm von M, wobei $\delta(q, a) = (q', b, wohin)$ durch das Wort

$$1^{q+1}01^{Zahl(a)}01^{q'+1}01^{Zahl(b)}01^{Zahl(wohin)}0$$

kodiert wird (mit Zahl(0)=1, Zahl(1)=2, Zahl(B)=3, Zahl(links)=1, Zahl(rechts)=2, Zahl(bleib)=3).
Die Simulation von M auf w funktioniert mit einer 3-Band-TM: (1) Gödelnummer von Band 1 löschen und auf Band 2 schreiben, (2) prüfen, ob $< M >$ syntaktisch korrekt ist, falls ja (3) Zustand 0 auf Band 3 schreiben und schrittweise Simulation durchführen, wobei immer Band

3 den gegenwärtigen Zustand speichert.
Für eine universelle TM U bedeutet dies:

M akzeptiert $w \Leftrightarrow U$ akzeptiert $< M > w$. M hält auf Eingabe $w \Leftrightarrow U$ hält auf Eingabe $< M > w$.

- **Reduzierbarkeit $\leq$:** L_1 und L_2 sind Sprachen über Σ_1 bzw. Σ_2. L_1 ist auf L_2 reduzierbar ($L_1 \leq L_2$) $\Leftrightarrow \exists$ eine stets haltenden TM T $\forall w \in \Sigma_1^* : w \in L_1 \Leftrightarrow T(w) \in L_2$.
  L sei unentscheidbar $\Rightarrow \overline{L}$ ist unentscheidbar und K ist unentscheidbar, wenn $L \leq K$.
  Aus $A \leq B$ folgt:

  - B rekursiv $\Rightarrow A$ rekursiv
  - A nicht rek. $\Rightarrow B$ nicht rek.

  Beispiel: $\overline{D} \leq U$.

  Diagonalsprache D: $D = \{< M > | M$ akzeptiert die Eingabe $< M >$ nicht. $\}$ ist nicht entscheidbar.

  Halteproblem: $H = \{< M > w | M$ hält auf Eingabe $w\}$ (nicht entscheidbar, Beweis mittels $U \leq H$)

  Spezielles Halteproblem: $H_\varepsilon = \{< M > | M$ hält auf dem leeren Wort $\varepsilon\}$ (nicht entscheidbar, Beweis mit mit $H \leq H_\varepsilon$)

  Universelle Sprache: $H = \{< M > w | M$ akzeptiert $w\}$ (nicht entscheidbar, Beweis mit $\overline{D} = \{w \in \{0,1\}^* | (w =< M >$ und M akzeptiert Eingabe $< M >)$ oder (w ist keine Gödelnummer)$\}$ und $\overline{D} \leq U$).
  Es gib also keinen Compiler, der stets korrekt voraussagt, ob ein Programm eine vorgelegte Eingabe akzeptiert.

- **Satz von Rice:** „Jede nicht-triviale Eigenschaft von TM führt zu einem nicht-trivialen Problem" oder: Sei f_M die (partielle) Funktion der TM M mit $F = \{f_M | M$ ist eine Turingmaschine$\}$ und $S \subseteq F$, dann ist die Sprache

$$TM(S) = \{< M > | M \text{ berechnet eine Funktion in } S\}$$

  nicht entscheidbar, falls $S \neq \emptyset$ und $S \neq F$ (Beweis mit $\overline{H_\varepsilon} \leq TM(S)$ über die überall undefinierte Funktion). Läßt sich sehr gut anwenden auf:

  - $L_1 = \{< M > | M$ häult für keine Eingabe$\}$
  - $L_2 = \{< M > | M$ gibt entweder stets die Ausgabe 0 oder 1$\}$
  - $L_3 = \{< M > | M$ berechnet die Identitätsfunktion$\}$
  - $L_4 = \{< M > | M$ akzeptiert alle Eingaben$\}$

- **Entscheidbarkeit:** L_1 und L_2 seien entscheidbar $\Rightarrow L_1 \cup L_2$, $L_1 \cap L_2$ und $\overline{L_1}$ entscheidbar

- **Rekursiv aufzählbar:** Die Sprache K heißt r.a. $\Leftrightarrow \exists$ TM M mit $K = L(M)$ (d.h. $w \in K \Leftrightarrow M$ akzeptiert w). Die TM M muß also nicht für jede Eingabe halten. Wenn w nicht zur Sprache gerhört, darf sie w niemals akzeptieren, muß es aber auch nicht erkennen.
  H, H_ε und U sind rekursiv aufzählbar aber nicht entscheidbar.
  L_1 und L_2 seien rekursiv aufzählbar $\Rightarrow L_1 \cup L_2$ und $L_1 \cap L_2$ rekursiv aufzählbar.
  L und $\overline{L}$ rek. aufz. $\Rightarrow L$ ist entscheidbar.
  $\overline{H}, \overline{H_\varepsilon}$ und $\overline{U}$ sind nicht rekursiv aufzählbar.

Sprache	Rekursiv	Rekursiv aufzählbar
H	$-\ (U \leq H)$	$+$
$\overline{H}$	$-$	$-$
H_ε	$-\ (H_l eq H_\varepsilon)$	$+$
$\overline{H_\varepsilon}$	$-$	$-$
U	$-$	$+$
$\overline{U}$	$-$	$-$
D	$-$	$-$
$\overline{D}$	$-\ (\overline{D} \leq U)$	$+\ (?)$

Für eine Sprache L gilt genau eine der folgenden 3 Eigenschaften:

1. L und $\overline{L}$ sind rek. (und damit auch rek. aufz.)
2. L und $\overline{L}$ sind nicht rek. aufz. (und damit auch nicht rek.)
3. Genau eine der beiden Sprachen L und $\overline{L}$ ist rek. aufz., aber nicht rek., die andere ist nicht rek. aufzählbar.

- **μ-rekursive Funktionen:** Neuer Formalismus, der wiederum das Konzept der turing-berechenbaren Funktionen liefert ($\rightarrow$ Church'sche These). Durch den vorgestellten Formalismus erhalten wir einen konstruktiven Beweis der Berechenbarkeit dieser neuen Klasse von Funktionen.
Gödelscher Unvollständigkeitssatz: *„Jede rekursiv aufzählbare Axiomatisierung der Zahlentheorie beitzt wahre, aber nicht beweisbare Aussagen."*. Ein Formalismus kann eine komplexere Realität also nicht exakt widerspiegeln.

Indikatorfunktion: oder charakteristische Funktion $f_L : \Sigma^* \rightarrow \{0,1\}$ von $L \subseteq \Sigma^*$ wenn gilt

$$\forall x \in \Sigma^* : x \in L \Leftrightarrow f(x) = 1$$

Z.B. $f(w) = 1 - (\Sigma_{i=1}^{|w|} w_i) \mod 2$ für die Menge der Worte mit gerader Anzahl von Einsen.

Grundfunktionen: Nachfolgefunktion: $S(x) = x+1$, Konstantenfunktion: $C_q^k(x_q, x_q, \ldots, x_k) = q$, Projektionsfunktion: $P_i^k(x_1, x_2, \ldots, x_i, \ldots, x_k) = x_i$

Einsetzung: Seien $\psi, \chi_1, \chi_2, \ldots, \chi_m$ Funktionen mit $\chi_i : \mathbb{N}^n \rightarrow \mathbb{N} \forall i$ und mit $\psi : \mathbb{N}^m \rightarrow \mathbb{N}$, so heißt $\rho : \mathbb{N}^n \rightarrow \mathbb{N}$ mit

$$\rho(x_1, x_2, \ldots, x_n) = \psi(\chi_1(x_1, x_2, \ldots, x_n), \ldots, \chi_m(x_1, x_2, \ldots, x_n))$$

die durch Einsatzung von $\chi_1, \chi_2, \ldots, \chi_m$ in ψ erhaltene Funktion.

Induktion: Ist $\chi : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$ und $\psi : \mathbb{N}^{n-1} \rightarrow \mathbb{N}$ Funktion, so ist $\rho : \mathbb{N}^n \rightarrow \mathbb{N}$ definiert als:

$$\begin{cases} \rho(0, x_2, \ldots, x_n) & = \psi(x_2, \ldots, x_n) \\ \rho(S(x_1), x_2, \ldots, x_n) & = \chi(x_1, \rho(x_1, x_2, \ldots, x_n), x_2, \ldots, x_n) \end{cases}$$

dir durch Indunktion aus χ mit Anfangsfunktion ψ hervorgegangene Funktion.

Primitiv Rekursiv: Eine Funktion heißt so, wenn sie durch endlich viele Anwendungen von Einsetzungen und Induktion basierend auf den Grundfunktionen darstellen läßt.
Ein Prädikat heißt primitiv rekursiv, wenn seine Indikatorfunktion primitiv rekursiv ist.
Jede primitiv rekursive Funktion ist berechenbar und jedes Prädikat (jede Sprache) mit primitiv rekursiver Indikatorfunktion ist entscheidbar.

Beispiele: Addition, Multiplikation, Potenzierung, Fakultät, Vorgängerfunktion, modifizierte Differenz, jede Minimums-/Maximumsfunktion, die Signum-Funktion, die Differenz, mod (x_1, x_2), $div(x_q, x_w$, die Prädikate gerade, ungerade, Teiler, vielfaches, $<$, $>$ etc., alle booleschen Ausdrücke

Endliche Summe über ψ: $\psi : \mathbb{N}^{n-+1} \to \mathbb{N}$ sei primitiv rekursiv, dann ist

$$\psi_\Sigma(z, x_1, \ldots, x_n) := \Sigma_{y<z}\psi(x_1, \ldots, x_n, y)$$

auch primitiv rekursiv.

Beschränkte Eigenschaften: Genauso gilt wenn A primitiv rekursives Prädikat auf sind $\mathbb{N}^n$ ist: $\exists_{y<x}A(x_1, \ldots, x_{n-1}, y)$ und $\forall_{y<x}A(x_1, \ldots, x_{n-1}, y)$ primitiv rekursive Prädikate auf $\mathbb{N}^n$.

Beschränkte eindeutige Existenzeigenschaft: A sie wieder primitiv rekursives Prädikat auf $\mathbb{N}^n$. Dann ist auch primitiv rekursiv $\exists^1_{y<z}A(x_1, \ldots, x_{n-1}, y)$

Es stellt keine Einschränkung dar, daß wir bei Turingmaschinen stets Eingabeworte haben, die aus Elementen eines endlichen Eingabealphabets zusammengesetzt sind, und bei p. r. Funktionen nur mit natürlichen Zahlen rechnen, da man diese Elemente in eine $(n+1)$-äre Darstellung (bei n Elementen im Eingabealphabet) verstehen kann.

Goldbachsche Vermutung: $\forall_{a>2}(a \text{ gerade}) : \exists_{x<a}x \text{ prim} \wedge (a \dot{-} x) \text{ prim}$

Beschränkter μ-Operator: Sei $A(x_1, \ldots, x_{n-1}, y)$ p.r. Prädikat. Dann ist

$$\mu_{y<z}A(x_1, \ldots, x_{n-1}, y) = \begin{cases} min\{y < x | A(x_1, \ldots, x_{n-1}, y)\} & \text{falls ein solches y existiert} \\ z & \text{sonst} \end{cases}$$

$\mu_{y<z}A(x_1, \ldots, x_{n-1}, y)$ ist primitiv rekursiv.

Ackermannfunktion: Die Funktion $a : \mathbb{N}^2 \to \mathbb{N}$ ist definiert als

$$\begin{aligned} a(0, y) &= y + 1 \\ a(x + 1, 0) &= a(x, 1) \\ a(x + 1, y + 1) &= a(x, a(x + 1, y)) \end{aligned}$$

mit den Eigenschaften:

1. $a(x, y) > x$
2. $a(x, y) > y$
3. Monotonie in x: $x_2 > x_1 \to a(x_2, y) \geq a(x_1, y)$
4. $a(x, y + 1) > a(x, y)$
5. Monotonie in y: $y_2 > y_1 \to a(x, y_2) > a(x, y_1)$
6. $a(x + 1, y) \geq a(x, y + 1)$
7. $a(x + 2, y) \geq a(x, 2y)$

Die Ackermannfunktion wächst schneller als jede primitiv rekursive Funktion. Formal: $f : \mathbb{N}^n \to \mathbb{N}$ primitiv rekursiv, dann gibt es einen Tempoparameter k, so daß

$$\forall_{\vec{x} \in \mathbb{N}^n} f(\vec{x}) < a(k, MAX\{x_1, \ldots, x_n\})$$

Die Ackermannfunktion ist primitiv rekursiv.

Unbeschränkter μ-Operator: Sei $A(x_1, \ldots, x_{n-1}, y)$ p.r. Prädikat. Dann ist

$$\mu A(x_1, \ldots, x_{n-1}, y) = \begin{cases} min\{y \in \mathbb{N} | A(x_1, \ldots, x_{n-1}, y)\} & \text{falls ein solches y existiert} \\ 0 & \text{sonst} \end{cases}$$

Durch Hinzunahme des unbeschränkten μ-Operator wird die Menge der primitiven μ-rekursiven Funktionen zur Menge der μ-rekursiven Funktionen erweitert. Die Menge der mit dem unbeschränkten μ-Operator entscheidbaren Prädikate entspricht genau den rekursiv aufzählbaren Sprachen.

Gödels Unvollständigkeitssatz: Nach der Definition der Formel ρ_L der Peano-Arithmetik für jede rekursiv aufzählbare Sprache L gilt, daß

$$x \in L \Leftrightarrow \rho_L(x) \text{ ist wahr}$$

Wenn nun Wahrheit und Beweisbarkeit übereinstimmen folgt also

$$x \in L \Leftrightarrow \rho_L(x) \text{ ist aus Peanos Axiomen ableitbar}$$

und damit

$$\begin{aligned} x \notin L &\Leftrightarrow \neg\rho_L(x) \text{ ist wahr} \\ &\Leftrightarrow (\neg\rho_L(x)) \text{ ist ableitbar} \end{aligned}$$

Damit wäre aber $\overline{L} = \{x|(\neg\rho_L(x)) \text{ ist ableitbar}\}$ und damit ist $\overline{L}$ rekursiv aufzählbar. Also ist für jede rekursiv aufzählbare Sprache L auch ihr Komplement rekursiv aufzählbar und jede rekursiv aufzählbare Sprache ist auch rekursiv, was ein Widerspruch wäre, da $\overline{D}$ rekursiv aufzählbar ist, wir aber die Nicht-Rekursivität nachgewiesen haben.

Probabilistische Turingmaschinen und Quantenrechner: Turingmaschinen und Registermaschinen sind gleichmächtig, wobei die Turingmaschinen sogar nur polynomiell langsamer sind. Auch parallele Registermaschinen können durch Turingmaschinen simuliert werden und zwar in Zeit $poly(n + t(n)) + p(n)$ bei Laufzeit von $t(n)$ der parallelelen RM $\rightarrow$ ein weiterer Grund für die Annahme, daß die Churchsche These richtig ist.
Nichtdeterministische Maschinenmodelle dienen nicht zur Rechenzeitmessung realer Rechner. Sie bilden ein Rechnermodell für die Klasse NP und unterstützen die komplexitätstheoretische Klassifikatione von Problemen.

Probabilistische TM: M wird beschrieben durch den Vektor

$$M = (Q, \Sigma, \delta, q_0, \Theta, F) \text{ bzw. } M = (Q, \Sigma, \delta, q_0, \Theta)$$

Dabei ist $\delta : \Theta \times Q \times \Theta \times Q \times \{links, rechts, bleib\} \rightarrow [0,1] \cap \mathbb{Q}$ und weißt jedem möglichen Übergang eine Wahrscheinlichkeit zu. Dabei gilt

$$\Sigma_{(\gamma', q', Richtung) \in \Theta \times Q \times \{links, bleib, rechts\}} \rho(\gamma, q, \delta', q', Richtung) = 1$$

Die von M akzeptierte Sprache ist dann $L_M = \{x \in \Sigma^* | p_x > \frac{1}{2}\}$.
Falls $\forall x \notin L \exists \varepsilon > 0 : p_x \leq \frac{1}{2} - \varepsilon$ und $\forall x \in L \exists \varepsilon > 0 : p_x \geq \frac{1}{2} + \varepsilon$, dann hat M einen **beschränkten Fehler**. Damit wird bei k-maligem Anwenden der Turingmaschine auf eine Eingabe x ein Fehler mit einer Wahrscheinlichkeit von höchstens $2^{-\Omega(k)}$ auftreten.

Quantenrechner: Die Struktur ähnlet der der probabilistischen Turingmaschine, jedoch ist δ wie folgt definiert:

$$\delta : \Theta \times Q \times \Theta \times Q \times \{links, bleib, rechts\} \rightarrow \mathbb{C} \cap \mathbb{Q} + i\mathbb{Q}$$

allerdings nur mit komplexen Zahlen der Länge von höchstens 1. Diesmal muß gelten

$$\sigma_{\gamma, q, \gamma', q', Richtung} |\rho(\gamma, q, \gamma', q', Richtung|^2 = 1$$

Wenn ein Quantenrechner Q in Zeit $t(n)$ rechnet, dann gibt es eine deterministische Turingmaschine M, die auf Platz $O(t^2(n))$ die Sprache L_Q erkennt. Hierzu ist aber die Forderung einesbeschränkten Fehlers ebenso notwendig wie die Forderung, daß die Konfigurationsmatrix A_Q unitär ist.

- Fragen:
- Beispiele: "berechenbar"? Wichtige Punkte!
- Definition: r.a. und entscheidbaren Sprachen

- Definition: entscheidbar
- Definition: H_ϵ und $\overline{H_\epsilon}$?
- Reduktion von $H \leq_p H_\epsilon$?
- Beispiel: Sprachen, die weder selbst noch ihr Komplement rekursiv aufzählbar sind? ($L = \{< M > | M \text{ hält immer}\}$)
- Beweis: $\overline{L}$ nicht r.a. (mit Reduktion $\overline{H_\epsilon} \leq \overline{L}$)
- Beispiel: rekursiv aufzählbare Sprachen! ($\{< M > | M \text{ hält auf mindestens einem} w \in \Sigma^*\}$)
- Beispiel: nicht rekursiv aufzählbare Sprachen!
- Halteproblem (Komplement auch r.a.? -Nein, weil sonst entscheidbar; Beweis: 2 Turingmaschinen verzahnt laufen lassen)
- Beweis: L r.a. und L nicht entscheidbar $\Rightarrow \overline{L}$ nicht r.a.
- Beispiel: Sprache, die weder rekursiv, noch r.a. ist?
- Beispiel: Sprache L, für die weder L noch $\overline{L}$ r.a. ist! ($\{< h > | h \text{ hält nicht}\}$)
- Satz von Rice
- Rekursionstheorem
- Church'sche These? Kann sie widerlegt werden (BILD)?
- Mächtigkeit der rekursiv aufzählbaren Sprachen?
- Reduktion m-Band TM auf 2 Band-TM
- Gödelscher Unvolständigkeitssatz

2.3 NP-vollständigkeit

Welche Probleme sind effizient berechenbar, welche nicht?
Wir unterscheiden 3 Varianten von Optimierungsproblemen:

- Maximierungs-/Minimierungsproblem: Berechnung einer Lösung deren Wert maximal/minimal ist.
- Zahlvariante: Es soll der Wert einer optimalen Lösung berechnet werden.
- Entscheidungsvariante: Für ein Limit soll entschieden werden, ob es oberhalb (bei Maximierungsproblemen) oder unterhalb (bei Minimierungsproblemen) der optimalen Lösung liegt.
- M

- **Komplexitätsklasse P:** Menge aller Sprachen L, die sich in polynomieller Zeit berechnen lassen: $L \in P \Leftrightarrow \exists$ (stets haltende) Turingmaschine M und eine Konstante k mit $L = L(M)$ und $Zeit_M(n) = O((n+1)^k)$ (die worst-case-Laufzeit ist also polynomiell in der Eingabelänge). L ist effizient berechenbar $\Leftrightarrow L \in P$ (Achtung: ist in allen Rechnermodellen dasselbe!).
  Ein Problem ist sicher nicht in P, wenn es nicht rekursiv ist, oder die Länge der Ausgabe nicht polynomiell beschränkt ist. Man kann aber auch zeigen, daß alle Algorithmen zur Problemlösung mehr als polynomielle Zeit benötigen, was aber sehr schwer ist.
  Beispiele:

  $Zh = \{code(G) | G \text{ ist ein ungerichteter, zusammenhängender Graph}\}$

  $Bipartit = \{code(G) | G \text{ ist ein ungerichteter, bipartiter Graph}\}$

  $Match = \{1^k 0 code(G) | G \text{ besitzt k Kanten, die keinen Endpunkt gemeinsam haben.}\}$

  Oder: $L \in NP \Rightarrow$ Es gibt eine (deterministische) TM M, mit $L = L(M)$ und $Zeit_M(n) \leq 2^{polynom(n)}$.

- **Komplexitätsklasse NP:** Probleme, die sie (wahrscheinlich) nicht effizient lösen lassen, aber effizient lösbar sind, wenn geraten werden darf, da sich dann eine richtige Lösung kurz und effizient verifizieren läßt.
  Die Sprache L gehört zur Komplexitätsklasse NP, wenn es eine nichtdeterministische TM M gibt mit $L = L(M)$ und $Zeit_M = O((n+1)^k)$ (bei Eingablängen n für eine natürliche Zahl k).

- **Nichtdeterministische TM:** $(Q, \Sigma, \delta, q_0, \Gamma, F)$, wobei $\delta \subseteq Q \times \Theta \times Q \times \Theta \times \{links, bleib, rechts\}$.
  Für eine gegebenen Zustand q und ein gelesenes Zeichen kann es also möglicherweise mehrere anwendbare Befehle geben, aus denen M auswählen kann.

  M **akzeptiert** $w \in \Sigma^*$: $\Leftrightarrow$ Es gibt eine Berechnung von M, die auf Eingabe w in einem akzeptierenden Zustand hält (umgekehrt genauso).

  Die von M akzeptierte Sprache: $L(M) = \{w \in \Sigma^* | M \text{ akzeptiert } w\}$

  Laufzeit: Worst-case: für Eingabelänge n das Maximum der Laufzeiten aller akzeptierten Eingaben der Länge n.
  $Schritte_M(w)$ =die minimale LAufzeit einer akzeptierenden Berechnung von M auf Eingabe w
  $Zeit_M(n) = max\{Schritte_M(w) | w \in \Sigma^n\}$

- **CLIQUEN-Problem:** Für einen ungerichteten Graphen G und eine Zahl $k \in \mathbb{N}$ ist gefragt, ob es eine Knotenteilmenge der Größe k von G gibt, in der je zwei Knoten durch eine Kante verbunden sind.

- **Vertex Cover-Problem:** $VC = \{code(G,k) | \text{Es gibt eine Knotenmenge Ü von höchstens k Knoten, so daß jede Kante mindestens einen Endpunkt in Ü besitzt}\}$

- **Set Cover:** $SC = \{code(A_1, \ldots, A_r, k) | \text{Es gibt höchstens } k \text{ Mengen } A_{i_1}, \ldots, A_{i_k} \text{ mit } \bigcup_{j=1}^{k} A_{i_j} = \bigcup_{l=1}^{r} A_l\}$.

- **Independent Set-Problem:** $IS = \{code(G,k) | \text{Es gibt eine Knotenmenge } U \text{ von mindestens } k \text{ Knoten, so daß keine zwei Knoten von } U \text{ durch eine Kante verbunden sind}\}$.

- **Null-Eins-Programmierproblem:** Eine Matrix A mit n Zeilen und m Spalten ist gegeben mit zwei Vektoren $b = (b_1, \ldots, b_n)$ und $c = (c_1, \ldots, c_m)$, sowie eine Zahl k uns es ist gefragt, ob ein $x \in \{0,1\}^m$ existiert mit $A \cdot x \leq b$ komponentenweise und $\Sigma_{i=1}^{m} c_i \cdot x_i \geq k$.
  Beim ganzzahligen Programmierproblem ist $x \in \mathbb{N}^m$.

- **Traveling Salesman Problem:** Vollständiger Graph $V_n = (\{1, \ldots, n\}, \{\{i,j\} | i \neq j\})$ mit n Knoten. Jede Kante e hat eine Länge $länge(e)$ und G hat eien Tour der Länge höchstens Bm falls es einen Kreis gibt, der jeden Knoten genau einmal furchläuft und dessen Summe der Kantenl2angen höchstens B ist.

- **Hamiltonsches Kreis-Problem:** Ungerichteter Graph G. Ein HC ist ein Kreis, der jeden Knoten genau einmal durcläuft, damit ist $HC = \{code(G) | G \text{ besitzt einen Hamiltonschen Kreis}\}$.
  Das Analoge Problem im gerichteten Graph wird DHC genannt.

- **Längste Wege:** $LW = \{code(G,k) | G \text{ besitzt einen Weg der Länge mindestens } k\}$.

- **MINIMUM-COVER-Problem:** Sei $M = \{M_1, \ldots, M_m\}$ mit $M_1 \ldots M_m \subseteq U$. Gibt es ein minimales $C \subseteq U$ mit $\bigcup_{M_i \in C} M_i = U$? Ebenfalls NP-volständig.

- **$P \subseteq NP$:** Eine deterministische TM ist auch eine nichtdeterministische TM.

- **Polynomiell reduzierbar:** $L_1 \subseteq \Sigma_1$ und $L_2 \subseteq \Sigma_2$ seien zwei Sprachen. L_1 ist auf L_2 reduzierbar (also: $L_1 \leq_p L_2$), wenn es eine TM M gibt mit:

  - M läuft in polynomieller Zeit

- M gibt für alle $w \in \Sigma_1^*$ die Ausgabe $M(w) \in \Sigma_2^*$, wobei gilt: $w \in \text{Ł}_1 \Leftrightarrow M(w) \in L_2$

Die Relation $\leq_p$ ist reflexiv und transitiv. Es gilt: $L_1 \leq_p L_2 \wedge L_2 \in P \Rightarrow L_1 \in P$.
Das Reduktionskonzept $\leq_p$ bezieht sich nur auf Entscheidungsprobleme, $\leq_T$ dagegen auf allgemeinere Probleme.

- **NP-vollständig:** L ist NP-vollständig $\Leftrightarrow L \in NP \wedge \forall K \in NP : K \leq_p L$

- **NP-hart:** L ist NP-hart $\Leftrightarrow \forall k \in NP : K \leq_p L$

- **Sätze:** L_1, L_2, L_3 und L Sprachen.

  1. $L_1 \leq_p L_2$ und $L_2 \in P \Rightarrow L_1 \in P$
  2. L ist NP-volständig und $P \neq NP \Rightarrow L \notin P$
  3. $L_1 \leq_p L_2$ und $L_2 \leq_p L_3 \Rightarrow L_1 \leq_p L_3$
  4. L_1 NP-vollständig und $L_1 \leq_p L_2 \Rightarrow L_2$ ist NP-hart. Wenn gilt $L_{\in} NP$, dann L_2 ist NP-vollst.

- **KNF-SAT:** $KNF{-}SAT = \{code(\alpha) | \alpha$ ist eine Formel in konjunktiver Normalform und α ist erfüllbar$\}$ also $\alpha \equiv k_1 \wedge k_2 \wedge \ldots \wedge k_r$ mit k_i Disjunktion von Literalen.

- **k-SAT:** $k - SAT = \{code(\alpha) \in KNF - SAT | \alpha$ besitzt höchstens k Literale pro Klausel$\}$ ist eine Abschwächung von $KNF - SAT$.

- **Praktische Beispiele: VLSI-Entwurf:** Das Problem der Schaltkreisminimierung ist NP-hart(weil der Algortihmus eine nucht erfüllbare Formel auf die 0 minimieren würde). Das Problem zu entscheiden, ob zwei vorgelegte Schaltkreise verschieden Funktionen berechnen sogar NP-vollständig (Vorgehen ähnlich, Vergleich mit 0).

  Ressourcenverteilung in Betriebssystemen: PARTITION-Problem: $\Sigma_{i \in I} t_i = \Sigma_{i \in \{1,\ldots,n\} \setminus I} t_i$
  SUBSET-SUM-Problem: $\Sigma_{i \in I} = Z$ wobei gilt: SUBSET-SUM$\leq_p$PARTITION, da man die Summe aller Werte von $T+1$ auf $2T+2$ erhöhen kann; dann ist eine Zerlegung in zwei Teilmengen der Größe $T+1$ möglich.

  Datenbanken (Schlüssel): Anwendung des MINIMUM-COVER-Problems: MINIMUM-COVER$\leq_p$KEY-FIND (Sei $U = \{u_1, \ldots, u_k\}$ und $M = \{M_1, \ldots, M_m\}$ und k die erlaubte Größe des Covers. Nun richte man für jede Menge in M ein Attribut ein und für jedes $u_i \in U$ zwei Datensätze d_i und d_i' mit Attribut n, wenn u_i nicht in M_n enthalten ist, ansonsten erhalten d_i und d_i' die Werte i bzw. $-i$.

  Gewinnstrategien bei Spielen: Finden der optimalen Gewinnstrategie:

$$F = \forall_{x_1} \exists_{x_2} \forall_{x_3} \exists_{x_4} \ldots \forall x_{n-1} \exists_{x_n} E(x_1, \ldots, x_n)$$

  wobe E ein Prädikat der Aussagenlogik ist, das wahr ist. Das Finden einer Belegung von F die zu einem wahren Ergebnis führt ist NP-hart.

- **NP-Vollständigkeits-Nachweise:** Beim Nachweis der NP-Vollständigkeit verwendet man verschiedene Strategien der polynomiellen Reduktion: die Restriktion, die lokale Ersetzung und die Transformation mit verbundenen Komponenten:

Restriktion	lokale Ersetzung	Transformation
SAT$\leq_p$SAT* (trivial)	SAT$\leq_p$3-SAT	3-SAT$\leq_p$CLIQUE
KP* $\leq_p$KP (trivial)	DHC$\leq_p$HC	3-SAT$\leq_p$DHC
PAR$\leq_p$BPP	3-SAT$\leq_p$KP*	
HC$\leq_p$TSP		
KP* $\leq_p$PAR		

Restriktion: Geeignet, um recht ähnliche Probleme aufeinander zu reduzieren, insbesondere, wenn das Problem, von dem die Restriktion ausgeht, ein (im weiteren Sinne) Spezialfall des anderen Problems ist.

Lokale Ersetzung: Wieder gut für ähnliche Probleme, besonders geeignet, wenn ein allgemeines Problem auf einen Spezialfall oder ein Spezialfall auf einen anderen reduziert werden soll.

Transformation: Geeignet, um verschiedenartige Probleme aufeinander zu reduzieren. Es werden häufig Komponenten mit sehr speziellen Eigenschaften benötigt. werden soll.

CLIQUE$\in$NP: Entwerfe nichtdet. 3-Band-TM M, Eingabe $w \in \{0,1\}^*$ auf Band 1.

1. M überprüft, ob $w = 1^k 01^n 0 code(G)$ (wobei $code(G)$ die Adjazenzmatrix des Graphen G zeilenweise hintereinander geschrieben $\rightarrow$ 0-1-Bitvektor der Länge n^2) also $|code(G)| = n^2 \rightarrow$ geht in Zeit polynomiell in n
2. M rät Knotenmenge und speichert zugehörigen Inzidenzenvektor auf Band 2 und Band 3 (dabei bleibt der erste Kopf stehen und die anderen beiden schreiben die (gleiche) Vermutung auf Band 2 und 3)
3. M verifiziert, ob geratene Knotenmenge einer Clique der Größe mind. k entspricht, prüft also, ob n Bits auf Band 2 geschrieben wurden, mindestens k Einsen geraten wurden und geratene Knotenmenge eine Clique ist (Kopf 1 läuft pro Takt einen Schritt nach rechts, Kopf 2 läuft alle n Takte einen Schritt nach rechts und Kopf 3 läuft pro Takt einen Schritt nach rechts, wird allerdings alle n Takte auf die $i - te$ Position zurück).

KNF-SAT ist NP-vollständig (Satz von Cook): Sei $L \in NP$, dann existiert eine nichtdeterministische TM M_L mit $L = L(M_L)$ und M_L hat polynomielle Laufzeit und es gibt für jedes Wort $w \in L$ eine akzeptierende Berechnung der Länge höchstens $polynom(|w|)$.
Sie $Q = \{0, q, \ldots, q\}$ und $\Gamma = \{0, 1, B\}$ mit 0 ist Anfangszustand und 1 der einzig akzeptierende Zustand und M_L hält stets, wenn Zustand 1 erreicht wird. Zu zeigen: Für jede Eingabe w kann eine Formel α_w in konjunktiver Normalform konstruiert werden, so daß: $\alpha_w \in$KNF-SAT $\Leftrightarrow$ Es gibt eien Berechnung von M_L, die w akzeptiert, wobei die Funktion $w \mapsto \alpha_w$ von einer transformierenden TM M in (deterministisch) polynomieller Zeit konstruierbar sein muß. Nun konstruieren wir eine Formel α, die alle Bedingungen an die TM M ausdrückt.
Der Beweis enthält zunächst eine Übertragung von nichtdeterministischen Turingmaschinen in den Rate-Verifikations-Modus und dann die Codierung polynomiell langer Rechenwege durch die Konjunktion polynomiell vieler in polynomieller Zeit berechenbarer Klauseln. Dabei sind die Klauseln genau dann gemeinsam erfüllbar, wenn es einen akzeptierenden Rechenweg der gegebenen TM gibt.
Die nichtdeterministische TM, die das Problem aus NP löst wird durch die Variablen des Boolschen Ausdrucks beschrieben: Die Kopfposition, der Bandinhalt an der aktuellen Position und der aktuelle Zustand.

KNF-SAT$\leq_p$3-SAT: Sei die Eingabe $\alpha \equiv k_1 \wedge \ldots \wedge k_r$ und M transformiert eine Klausel k_i in eine Konjunktion $k_{i,1} \wedge \ldots \wedge k_{i,r_i} \equiv \alpha_i$ von Klauseln mit höchstens drei Literalen, wobei k_i wenn $k_i = y_1 \vee \ldots \vee y_l$ mehr als 3 Literale besitzt in neue Klauseln umgeformt wird:

$$
\begin{aligned}
k_{i,1} &\equiv y_1 \vee y_2 \vee z_{i,1} \\
k_{i,2} &\equiv \neg z_{i,1} \vee y_3 \vee z_{i,2} \\
k_{i,3} &\equiv \neg z_{i,2} \vee y_4 \vee z_{i,3} \\
&\vdots \\
k_{i,l-3} &\equiv \neg z_{i,l-4} \vee y_{l-2} \vee z_{i,l-3} \\
k_{i,l-2} &\equiv \neg z_{i,l-3} \vee y_{l-1} \vee y_l
\end{aligned}
$$

Damit sind alle Klauseln $k_{i,1}, \ldots, k_{i,l-2}$ wahr für $z_{i,1} = \ldots = z_{i,j-2} = wahr$ und $z_{i,j-1} = \ldots = z_{i,l-3} = falsch$.
Also ist

$\alpha \in KNF - SAT$
$\Leftrightarrow$Es existiert eine Belegung, die jede der Klauseln $k_1, \ldots, k_r$ wahr macht.
$\Leftrightarrow$Es gibt eine Belegung, die alle Formeln $\alpha_1, \ldots, \alpha_r$ wahr macht.
$\Leftrightarrow M(\alpha) \in 3 - SAT$

3-SAT$\leq_p$CLIQUE: TM M konstruieren, so daß gilt:

$$\alpha \in 3 - SAT \Leftrightarrow M(\alpha) \in CLIQUE$$

Sei $\alpha \equiv k_1 \wedge \ldots \wedge k_r$ mit $k_i \equiv l_{i,1} \vee l_{i,2} \vee l_{i,3}$, wobei M der Formel α einen Graphen $G(\alpha)$ zuweist, der für jede Klausel k_i eine Gruppe $g_i = \{(i,1),(i,2),(i,3)\}$ von 3 Knoten besitzt und damit $\bigcup_{i=1}^{r} g_i = V$. Die Knoten einer Gruppe sind nicht mit Kanten verbunden und wir verbinden die Knoten (i,r) und (j,s) mit $i \neq j$ nur dann nicht, wenn $l_{i,r} \equiv \neg l_{j,s}$ (wenn also $l_{i,r}$ und $l_{j,s}$ nicht beide simultan erfüllbar sind). Es ist also $M(\alpha) := (r, 3r, G(\alpha))$

CLIQUE$\leq_p$IS: Wenn $G = (V,E)$, dann ist $\overline{G} = (V, \overline{E})$ mit $\overline{E} = \{\{i,j\} | i \neq j \text{ und } \{i,j\} \notin E\}$ und damit: C ist CLIQUE in $G \Leftrightarrow C$ ist unabhängige Menge für $\overline{G}$, also $M((G,k)) = (\overline{G}, k)$

IS$\leq_p$VC: I ist unabhängige Menge $\Leftrightarrow V \setminus I$ ist Knotenüberdeckung. und somit $M((G,k)) = (G, |V| - k)$.

VC$\leq_p$SC: Sei (G,k) die Eingabe und $A_i = \{e \in E | e \text{ besitzt i als Endpunkt}\}$. Dann ist: $\ddot{U} \subseteq V$ Knotenüberdeckung $\Leftrightarrow \forall e \in E$ besitzt Endpunkt in $\ddot{U} \Leftrightarrow \bigcup_{i \in \ddot{U}} A_i = \bigcup_{j=1}^{n} A_j = E$. Also $M((G,k)) = (A_1, \ldots, A_n, k)$

KNF-SAT$\leq_p$Null-Eins-Programmierung: Sei $\alpha \equiv k_1 \wedge \ldots \wedge k_t$ mit $k_i = x_{i_1} \vee x_{i_2} \vee \ldots \vee x_{i_m}$ mit $x_{i_j} \in \{b_1, \overline{b_1}, \ldots, b_n, \overline{b_n}\}$ und wir transformieren ein Null-Eins-Programm mittels $\Phi(code(\alpha)) = (A,b,c,k)$. Jede Klausel k_i der Formel α wird umschrieben durch $\Sigma_{i \in J} x_i + \Sigma_{i \notin J}(1 - x_i) \geq 1$ mit $J \subseteq \{1, \ldots, n\}$ die Menge der Variablen, die nicht negiert in k_i vorkommen. Aus den Gleichungen$(*)_i$ mit $1 \leq i \leq t$ für alle Klauseln erhält man die Matrix $-A$, der Vektor $b = -I = (-1, \ldots, -1)$. Für die Bildung von c und k kann man einfach die Ungleichung für k_1 nehmen und von dieser c^T und k ableiten.

Null-Eins-Programmierung$\leq_p$Ganzzahlige Programmierung: ?

DHC$\leq_p$HC: Aus dem gerichteten Graphen $G = (V,E)$ konstruieren wir einen ungerichteten Graphen $G' = (V', E')$, so daß G einen HC besitzt $\Leftrightarrow G'$ einen besitzt. Dazu spalten wir jeden Knoten v in G in drei Knoten auf: $(v, ein), (v, mitte), (v, aus)$.

HC$\leq_p$TSP: Für eine Eingabe G mit Knotenmenge $V = \{1, \ldots, n\}$ wählen wir den vollständigen Graphen V_n und definieren

$$länge(\{i,j\}) = \begin{cases} 1 & \text{wenn } \{i,j\} \in E \\ 2 & \text{sonst} \end{cases}$$

mit $B = n$. Somit ist $M(G) = (V_n, länge, B)$

HC$\leq_p$LW: Sei G ungerichtet, $V = \{1, \ldots, n\}$. G besitzt genau dann einen HC, wenn G einen Weg der Länge $n-1$ besitzt, der in Knoten 1 beginnt und mit Nachbarn vom Knoten 1 endet. Jetzt erfinden wir einen neuen Knoten $1'$, den wir zu G hinzufügen und verbinden ihn mit allen Nachbarn von 1. Dann erfinden wir zwei weitere Knoten 0 und $0'$ und verbinden 0 mit 1 und $0'$ mit $1'$. Sie nun $G' = \{0, 1, \ldots, n, 0', 1'\}$, dann folgt

$G \in HC \Leftrightarrow$ G hat einen Weg der Länge n-1, der in 1 beginnt und mit einem Nachbarn von 1 endet
$\Leftrightarrow$ G' hat einen Weg der Länge n+2

3-SAT$\leq_p$DHC: Gegeben ist die Fromel $\alpha_i \equiv l_{i,1} \vee l_{i,2} \vee l_{i,3}$ mit $\alpha_i \equiv l_{i,1} \vee l_{i,2} \vee l_{i,3}$, Dabei sei angenommen, daß genau die Variablen $X_1, \ldots, x_n$ (bzw. ihre Negation) in α vorkommen. Nun konstruieren wir den gerichteten Graphen $G(\alpha)$ in polynomieller Zeit, so daß gilt: α ist erfüllbar $\Leftrightarrow$ $G(\alpha)$ besitzt einen Hamiltonschen Kreis.
$G(\alpha)$ besteht aus den Variablenknoten $x_1, \ldots, x_n$ und r Teilgraphen $G_1, \ldots, G_r$, wobei G_i der Klausel α_i entspricht. Jett fügren wir die Knoten wie folgt ein:
Alle x_i haben zwei ausgehende Kanten (pos. und neg.). Die positive bzw. negative Kante wird auf die erste Klausel gesetzt, in der x_i bzw. $\neg x_i$ vorkommt.
Dabei nehmen wir die Kante des ersten, zweiten oder dritten Literals, je nachdem, an welcher Stelle x_i vorkommt und verbinden sie mit x_i, dann verbinden wir $1'/2'/3'$ mit der nächsten Klausel, in der x_i vorkommt. Gibt es so eine nicht mehr, so verbinden wir die Kante mit x_{i+1}.
G besitzt also $n + 6m$ Knoten.

3SAT$\leq_p$SUBSET-SUM: Sei $F = K_1 \wedge K_2 \wedge \ldots \wedge K_m$ mit $K_i = (l_{i,1} \vee l_{i,2} \vee l_{i,3})$ mit $l_{i,j} \in \{x_1, \overline{x_1}, x_2, \ldots, x_n, \overline{x_n}\}$. Nun konstruiere $Z = \underbrace{44\ldots4}_{m-mal}\underbrace{11\ldots1}_{n-mal}$. Die Menge I mit den Zahlen, über denen die Summe gebildet werden soll besteht aus $pos_1, \ldots, pos_n$ (Literal x_i: im linken, m Stellen breiten Ziffernblock enthält pos_i an Stelle j eine 1, falls x_i in Klausel j auftaucht; im rechten Ziffernblock steht an Position i eine 1), $neg_1, \ldots, neg_n$ (gleiches wie bei pos_i nur für $\overline{x_i}$), $small_1, \ldots, small_m$ ($small_i$ enthält an Stelle i im ersten Ziffernblock eine 1) und $big_1, \ldots, big_m$ ($big_i := 2 \cdot small_i$).
Also sind PARTITION und SUBSET-SUM NP-vollständig.

Zusammenfassung:

$$SAT \begin{cases} \leq_p^1 & SAT^* \\ \leq_p^2 & 3-SAT \begin{cases} \leq_p^3 DHC \leq_p^2 HC \leq_p^1 TSP \\ \leq_p^3 CLIQUE \\ \leq_p^{2?} KP \begin{cases} \leq_p^1 KP \\ \leq_p^{1?} PAR \leq_p^1 BPP \end{cases} \end{cases} \end{cases}$$

- Fragen:
- Was ist NP?
- Wann ist eine Sprache NP-vollst.?
- Gibt es "vernünftige" Probleme $\notin NP$? (z.B. Ist eine gegebene Zugfolge in einem Spiel optimal?)
- Polynomielle Reduktion $\leq_p$ (formal, detailliert)
- Sind alle Probleme in NP-entscheidbar? Ja, weil NDTM in DTM umwandelbar
- Polynomielle Reduktion 3-SAT auf CLIQUE
- Polynomielle Reduktion CLIQUE auf SAT
- Satz von Cook (Beweisidee)
- Was ist SAT?
- Ist KNF-SAT NP-vollständig? (Grundidee)
- Gibt es kontextfreie Sprachen, die NP-vollst. sind? (nein ,wegen CYK)
- 4-SAT$\leq_P$CLIQUE
- Gibt es eine kontextfreie Sprache, die NP-vollst. ist?
- Wie kann man das TSP apporximieren?
- Reduktion HC auf TSP

2.4 Endliche Automaten

Endliche Automaten bilden für viele praxisrelevante Probleme das passende Modell. Alle Schaltwerke lassen sich durch endliche Automaten darstellen und schließlich bilden endliche Automaten Modelle für VLSI-Chips und ganze Rechner ohne virtuellen Speicher. Dabei sind Anzahl der Zustände und Speicher austauschbare Begriffe (wenn k Zustände ausreichen, genügt eine Speicher, der k Informationen aufnehmen kann).

- **Endlicher Automat:** $A = (Q, \Sigma, \delta, q_0, F)$ mit Eingabealphabet Σ, endliche Menge Q von Zuständen, Anfangszustand q_0, Menge F der akzeptierenden Zustände, Programm $\delta : Q \times \Sigma \to Q$.
  A liest nun eine Wort $w \in \Sigma^*$ von links nach rechts und wechselt seine Zustände gemäß δ. Bezogen auf Wort ist also $\delta : Q \times \Sigma^* \to Q$ und $\delta(q, \varepsilon) = q$ und $\delta(q, w_1 \dots w_{n+1}) = \delta(\delta(q, w_1 \dots w_n), w_{n+1})$.
  A akzeptiert $w \in \Sigma^* \Leftrightarrow \delta(q_0, w) \in F$, wobei $L(A) = \{w \in \Sigma^* | A \text{ akzeptiert } w\}$ die von A akzeptierte Sprache ist.
  $L \subseteq \Sigma^*$ heißt **reguläre Sprache** $\Leftrightarrow \exists$ Automat A mit $L = L(A)$.
  Der Automat läßt sich am besten durch ein Zustandsdiagramm verdeutlichen mit Knoten q_i und Kanten (q, q'), wenn $\delta(q, a) = q'$, und der Kantenbeschriftung a.

- **Äquivalente Zustände:** Zwei Zustände $p, q \in Q$ heißen äquivalent ($p \equiv_A q$) $\Leftrightarrow \forall w \in \Sigma^*$ gilt: $\delta(p, q) \in F \Leftrightarrow \delta(q, w) \in F$
  Dabei ist $\equiv_A$ eine Äquivalenzrelation.

- **Äquivalenzklassen:** Mit $A = (Q, \Sigma, \delta, q_0, F)$ sei für $p \in Q$: $[P] = \{q \in Q | p \equiv_A q\}$ die **Äquivalenzklasse** von p.
  Der **Äquivalenzklassenautomat** A' von A besitzt $Q = \{[p] | p \in Q\}$ mit Anfangszustand $[q_0]$ und $F' = \{[p] | p \in F\}$. Dabei ist $\delta'([p], a) = [\delta(p, a)]$. A' ist wohldefiniert und äquivalent zu A.
  Oft ist es einfacher zu zeigen, daß zwei Zustände nicht äquivalent sind anstatt umgekehrt. Dafür kann man sog. **Zeugen** $w \in \Sigma^*$ finden.

- **Bestimmung äquivalenter Zustände:** Man konstruiere eine Adjazenzlistendarstellung von G und markiere alle Paare $\{p, q\}$ mit $p \in F, q \notin F$ oder $p \notin F, q \in F$. Führe dann eine Tiefensuche an den markierten Knoten durch, die alle besuchten Knoten markiert.
  Die Laufzeit ist dann $O(|Q|^2 \cdot |\Sigma|)$ (Korrektheit mit kürzestem Zeugen und Laufzeit mit Initialisierung $O(|Q|^2 \cdot |\Sigma|)$, weil G $\binom{|Q|}{2} = \Theta(|Q|^2)$ Knoten und höchstens $\binom{|Q|}{2} \cdot |\Sigma|$ Kanten besitzt und dehalb die Tiefensuche maximal Zeit $= (|Q|^2 + |Q|^2 \cdot |\Sigma|) = O(|Q|^2 \cdot |\Sigma|)$ benötigt).

- **Nerode-Relation $\equiv_L$:** L sei Sprache über Σ. Die Nerode-Relation $\equiv_L$ für Worte in Σ^* ist wie folgt definiert:
  $$x \equiv_L y \Leftrightarrow (\forall w \in \Sigma^* : xw \in L \Leftrightarrow yw \in L)$$
  Mit dem **Index von** L definieren wir die Anzahl der Äquivalenzklassen von $\equiv_L$. Er beschreibt die Anzahl der Zustände des minimalen endlichen Automaten, den wir aus der Nerode-Relation bauen können.

- **Äquivalenzrelation R_A:** R_A ist eine rechtsinvariante Äquivalenzrelation (also: $uR_Av \Rightarrow uzR_Avz \forall x \in \Sigma^*$ und reflexiv, transitiv, symmetrisch).
  Sei $L = L(A)$, dann ist $xR_Ay \Leftrightarrow \delta(q_0, x) = \delta(q_0, y)$ mit q_0 Anfangszustand von A. Dann:
  - entsprechen die Äquivalenzklassen von R_A den nicht überflüssigen Zuständen von A.
  - gilt: $xR_Ay \Rightarrow x \equiv_L y$
  - gilt mit der Zustandsmenge Q von A: $|Q| \geq$Index von L

- **Nerode-Automat N_L für L:**
  - Die Zustände vonN_L sind Äquivalenzklassen von $\equiv_L$
  - Anfangszustand ist $[\varepsilon]$

- Akzeptierende Zustände: $F = \{[x] | x \in L\}$
- Programm: $\delta([x], a) = [xa]$

$L = L(N_L)$ und N_L ist minimal: Jeder Automat, der L akzeptiert hat mindestens so viele Zustände wie N_L. Dabei sind der Äquivalenzklassenautomat und der Nerode-Automat isomorph.

- Satz von Nerode: L regulär $\Leftrightarrow$ Der Index von L ist endlich.
Damit ist dies eine hinreichende Bedingung für die Regularität der Sprache L.

- Minimierung endlicher Automaten: Sei $A = (Q, \Sigma, \delta, q_0, F)$.

1. Alle Zustände, die von q_0 nicht erreicht werden können sind überflüssig und können entfernt werden (Tiefensuche von q_0 durchführen).
2. Konstruiere den Äquivalenzklassenautomat (Markierungsalgorithmus) in Zeit $O(|Q|^2 \cdot |\Sigma|)$. Dabei bilden wir alle möglichen Paare von Zuständen und markieren zunächst die Paare, die akzeptierende und nicht akzeptierende Zustände beinhalten. Dann betrachten wir Zustände mit Zeugen der Länge 1 und markieren die Paare, die Zustände enthalten, die mit der Eingabe eines Buchstabens in einen akzeptierenden bzw. nicht akzeptierenden Zustand kommen usw. Die am Ende nicht markierten Paare bilden dann Äquivalenzklassen.

Die Minimierungen, die vorgenommen werden sind naheliegend. Erst der Satz von Nerode belegt, daß der Automat auch minimal ist.

- Pumping-Lemma: Wann ist eine Sprache nicht reguäre (sprengt also die Grenzen der Berechnungskraft endlicher Automaten)?
Sei L reguläre. Dann ex. eine Pumpingkonstante n, so daß jedes Wort $z \in L$ mit $|Z| \geq n$ eine Zerlegung mit den folgenden Eigenschaften besitzt:

- $z = uvw$, $|uv| \leq n$ und $|v| \geq 1$
- $uv^i w \in L$ für jedes $i \geq 0$

Wenn also die Worte der Sprache lang genug sind, dann gibt es ein nicht-leeres Teilwort v, das „aufgepumpt" ($i \geq 1$) oder „abgepumpt" ($i = 0$) werden kann.) $\rightarrow$ Beweis mit endlichem Automat A und Pumpingkonstante $n = |Q|$ und $z \in L$ mit $z = z_1 z_2 \ldots z_s$ und $|z| = s \geq n$ beliebig, also muß bei der Erkennung von z ein Schleife durchlaufen worden sein.
Um zu zeigen, daß eine Sprache L nicht regulär ist, müssen wir das Pumping-Lemma falsifizieren:

- $\exists$(unbekannte) Pumpingkonstantre n
- Wir wählen „geeignetes Wort" $z \in L$ mit $|z| \geq n$
- Für z gibt es eine Zerlegung $z = uvw$ ($|uv| \leq n$, $|v| \geq 1$), so daß $uv^i w \in L$. Wir müssen also für jede mögliche Zerlegung $z = uvw$ ein i finden, so daß $uv^i w \notin L$.

Formal gilt also:

$$L \text{ regulär} \Rightarrow \exists N \in \mathbb{N} \forall z \in L, |z| \leq N \exists \text{Zerlegung } z = uvw, |uv| \leq N, |v| \geq 1 \forall i \geq 0 : uv^i w \in L$$

Da das Pumping-Lemma eine notwendige, aber nicht hinreichende Bedingung für die Regularität von L ist, läßt sich die nicht-Regularität von L nachweisen, durch:

$$\forall N \in \mathbb{N} \exists z \in L, |z| \geq N \forall \text{Zerlegung } z = uvw, |uv| \leq N, |v| \geq 1 \exists i \geq 0 : uv^i w \notin L \Rightarrow L \text{ nicht regulär}$$

Gutes Beispiel: $L = \{0^n 1^n | n \in \mathbb{N}\}$ mit $N = n$.

- **Nichtdeterministischer endlicher Automat:** $A = (Q, \Sigma, \delta, q_0, F)$ mit $\delta : Q \times \Sigma \to P(Q)$, wobei $P(Q)$ die Potenzmenge von Q ist.
  Das Programm eines nichtdeterministischen Automaten weist also jeder „Konfiguration" (p, a) von Zustand und Buchstaben eine Menge von Zuständen zu.

  Fortsetzung: $\delta : Q \times \Sigma^* \to P(Q)$ sie die Fortsetzung con δ mit: $\delta(q, \varepsilon) = \{q\}$ und $\delta(q, wa) = \bigcup_{p \in \delta(q,w)} \delta(p, a)$ (der Zustand $r \in Q$ ist also durch Eingabe wa vom Zustand q aus erreichbar, falls ein Zustand p durch Eingabe w vom Zustand q erreichbar ist und $r \in \delta(p, a)$.

  N **akzeptiert** w: $\Leftrightarrow \exists p \in F$ mit $p \in \delta(q_0, w)$

  Die von N **akzeptierte Sprache:** $L(N) = \{w \in \Sigma^* | N \text{ akzeptiert } w\}$

  Es kann sein, daß man für einen DFA 2^k Zustände benötigt und die gleiche Sprache mit einem NFA und $k + 1$ Zuständen darstellen kann (z.B. bei dem Automat der die Sprache erkennt, deren Wörter als k-t letzten Buchstaben eine 1 haben) . NFA's können in DFA umgebaut werden, die erkannte Sprache ist also auch regulär: Wir simulieren dabei alle Berechnungen von N gleichzeitig und wählen Teilmengen aus Q als Zustände. Ein Zustand $s \subseteq Q$ ist dann

  $$s = \{p \in Q | \text{Es gibt eine Berechnung von N für w, die in p endet}\} = \delta_N(q_0, w)$$

  NFA sind einfacher bei den Operationen Vereinigung, Konkatenation und Kleenescher Abschluß einsetzbar. Diese Operationen sind implizit nichtdeterminitisch defininiert, da sie einen Existenzquantoren enthalten.

- **DFA $\Leftrightarrow$ NFA:** Sei N NFA $A = (Q_N, \Sigma, \delta_N, q_0, F_N)$. Der DFA A hat dann Komponenten:

  - $Q_A = \{t | t \subseteq Q_N\}$
  - Anfangszustand $\{q_0\}$
  - $F_A = \{t \in Q_A | t \text{ enthält einen Zustand aus } F_N\}$
  - δ_A ist definiert durch $\delta_A = (t, a) = \{p \in Q_N | \text{Es gibt } q \in t \text{ mit } p \in \delta_N(q, a)\}$

  Dann sind N und A äquivalent, d.h. $L(N) = L(A)$. Aus einem NFA kann also ein DFA mittels Potenzmengenkonstruktion erstellt werden.

- **NFA mit ε-Übergängen:** ist wie ein DFA definiert, nur daß $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to P(Q)$. Diese Eigenschaft hat keinen Einfluß auf die Regularität der erkannten Sprache.

- **Leerheitstest:** Test, ob ein akzeptierender Zustand erreichbar ist.

- **Vollständigkeitstest:** Test, ob alle Worte entschieden werden. Dabei genügt es nicht zu zeigen, daß ein nichtakzeptierender Zustand erreicht wird, da eine andere Berechnung zu einem akz. zustand führen kann. Der Test ist bei NFA's also NP-hart.

- **Gleichheitstest:** Es soll entschieden werden, ob zwei endliche Automaten A_1 und A_2 die gleiche Sprache akzeptieren. Das kann in Zeit $O(|Q_1||Q_2||\Sigma|)$ erreicht werden, da es genügt zu zeigen, daß $L_1 \cap \overline{L_2} = \emptyset$ und $\overline{L_1} \cap L_2 = \emptyset$ ist.

- **Regularität:** Seien L, L_1 und L_2 regulär, dann sind regulär: $\overline{L}$, $L_1 \cup L_2$ und $L_1 \cap L_2$, $L_1 \circ L_2$ und L^*.
  Reguläre Sprachen sind also unter den Operationen textitVereinigung, Durchschnitt, Komplement, Konkatenation, Kleene-Abschluß abgeschlossen.

- **Reguläre Ausdrücke:** Beschreibungsschema für reguläre Sprachen mit Alphabet $\Sigma = \{a_1, \ldots, a_k\}$:

  1. $\emptyset, \varepsilon, a_1, \ldots, a_k$ sind reguläre Ausdrücke
  2. Wenn R r.A., dann ist R^* und (R) r.A.
  3. Wenn R_1 und R_2 r.A., dann auch $R_1 + R_2$ und $R_1 \cdot R_2$

Wenn R ein regulärer Ausdruck für eine Sprache L ist, dann ist L regulär.
Wenn L regulär ist über Σ, dann gibt es einen r.A. für L (Beweis mit dynamischem Programmieren).

- Grammatik: $G = (\Sigma, V, S, P)$ mit

- Endliches Alphabet Σ
- Endliche Menge V von Variabalen mit $\Sigma \cap V = \emptyset$
- Startsymbol $S \in V$
- Endliche Menge P von Produktionen der Form (u, v) mit $u \in (\Sigma \cup V)^+)$ und $v \in (\Sigma \cup V)^*$
  Sei (u, v) eine Produktion von G, dann definieren wir die Worte $w_1, w_2 \in (\Sigma \cup V)^*$:

$$w_1 \to^* w_2 \Leftrightarrow \text{Es gibt } x, y \in (\Sigma \cup V)^* \text{ mit } w_1 = xuy \text{ und } w_2 = xvy$$

  Seien $r, s \in (\Sigma \cup V)^*$, dann definieren wir:

$$r \to s \Leftrightarrow \text{Es gibt Worte } w_1 = r, w_2, \ldots, w_k = s, \text{ so daß } w_1 \to w_2 \to \ldots \to w_k$$

  . $L(G) = \{w \in \Sigma^* | S \to^* w\}$ ist die **von der Grammatik erzeugte Sprache**.

Eine Grammatik heißt **regulär**, wenn alle Produktionen P die Form haben:

$$u \to \varepsilon (\text{für } u \in V) \text{ oder}$$
$$u \to av \text{für } u, v \in V, a \in \Sigma)$$

L regulär $\Leftrightarrow$ Es gibt eine reguläre Grammatik G mit $L = L(G)$.

- Fragen:

- Zusammenhang regulärer Ausdruck - reguläre Grammatik
- Definition: Reguläre Sprachen über DFA
- Potenzmengenkonstruktion
- Pumping Lemma
- Äquivalenz-Automat
- Algorithmus: DEA (DFA) $\to$ Reg. Ausdruck (Idee)
- ist $L = \{\omega\omega | \omega \in a, b^*\}$ regulär, kontextfrei oder kontextsensitiv?
- Wie finde ich den Minimalautomat?
- Isomorphie von Automaten?
- Nerode-Relation $\equiv_L$ und -automat
  - Was ist Menge der Zustände?
  - Wie sieht ein Programm aus?
  - Was ist der Index L?
  - Äquivalenzklassen?
  - N_L
  - Minimaler Automat? xR_Ly Beweis; Algorithmus
  - Index von $\{a^n b^n | n \in \mathbb{N}\}$?
  - Wie wählt man den Index für $\{a^n b^n | n \in \mathbb{N}\}$? ($\infty$, da nicht regulär!)
  - Beweis: Index für $L = \{a^n b^m c^{n+m}\}$ ist ∞, da L nicht reulär
- Aus einem Automat einen regulären Ausdruck konstruieren
- Grammatiken
- Kontextsensitive Sprachen (warum linear platzbeschränkt?)
- Sind regulären Sprachen in P enthalten?
- Wie kann aus einem DFA ein regulärer Ausdruck für die von ihm erzeugte Sprache erzeugt werden?
- Welche Sprachen sind r.a., in P, in NP?

2.5 Kontextfreie Sprachen

Z'iel ist die Beschreibung von Programmiersprachen durch formale Grammatiken (die einerseits ausdrucksstark und andererseits effiziente Lösungen des Compiler-Problems erlauben).

- **Chomsky-Hierarchie: Typ-0-Grammatiken:** Alle Grammatiken ohne Einschränkungen: $L_0 = \{L(G)|G \text{ ist vom Typ } 0\}$
  Die Klasse der Typ-0-Grammatiken stimmt mit der Klasse der rekursiv aufzählbaren Funktionen überein, läßt sich also von einer nichtdeterministischen Turingmaschine in polynomieller Zeit erkennen (bei der Simulation durch eine determinstische TM simulieren wir für ein wachsendes l alle möglichen Rechenwege hintereinander).

  Kontextsensitiv (Typ 1): Produktionen der Form $u \to v$ mit $|u| \leq |v|$: $L_1 = \{L(G)|G \text{ ist vom Typ } 1\} \cup \{L(G) \cup \{\varepsilon\}|G \text{ ist vom Typ } 1\}$
  Die rechte Seite jeder Ableitung darf nicht kürzer sein als die linke. Es darf lediglich die Produktionsregel $S \to \varepsilon$ geben, wobei jedoch S niemals auf der rechten Seite stehen darf. Wörter dürfen also nicht verkürzt werden!
  Die Klasse der kontextsensitiven Sprachen stimmt mit der Klasse der von nichtdeterministischen TM auf linearem Platz $|w|$ akzeptierten Wörter w überein. Da es rekursiv aufzählbare, aber nicht kontextsensitive Sprachen gibt, gibt es auch rekursiv aufzählbare, aber nicht kontextfreie Sprachen. Die Klasse der Kontextsensitiven Sprachen ist noch zu allgemein, um das WOrtproblem effizient zu lösen, da dieses zwar rekursiv, aber dennoch NP-hart ist (Beispiel: das Erfüllbarkeitsproblem SAT läßt sich sogar mit einer det. TM lösen, dennoch ist bereits die Grammatik, die SAT erzeugt, NP-vollständig!).

  Kontextfrei (Typ 2): Produktionen der Form $u \to v$ mit $u \in V$ und $v \in (V \cup \Sigma)^*$: $L_2 = \{L(G)|G \text{ hat Typ } 2\}$

  Reguläre Grammatik (Typ 3): $L_3 = \{L(G)|G \text{ hat Typ } 3\}$

  L_0 ist die Klasse aller rekursiv aufzählbaren Sprachen.
  L_1 ist die Klasse aller Sprachen, die von nichtdeterministischen Turingmaschinen auf linearem Platz erkannt werden. Jede Sprache aus L_1 ist entscheidbar.
  $L_3 \subset L_2 \subset L_1 \subset L_0$.
  Es ergeben sich folgende Beziehungen bei den Ableitungen:

$\to$	DEA	NEA	RG	RA
DEA	–	linear	linear	exp.
NEA	exp.	–	linear	exp.
RG	exp.	linear	–	exp.
RA	exp.	linear	linear	–

  Reguläre Grammatiiken und nichtdeterministische endliche Automaten sind gleich kompakte Beschreibungsformen der Klasse regulärer Sprachen. Reguläre Ausdrücke und deterministische endliche Automaten können exponentiell länger, aber nicht kürzer sein. Auch zwischen regulären Ausdrücken und deterministischen endlichen Automaten gibt es exponentielle Blow-ups.

- **DTAPE(s(n)) und NTAPE(S(n)):** Klassen der Sprachen, die von einer deterministischen bzw. nichtdeterministischen TM mit Platzbedarf $s(n)$ akzeptiert werden können.

- **PSPACE und NSPACE:** Vereinigung aller DTAPE(n^k), $k \in \mathbb{N}$ und Vereinigung aller NTAPE(n^k), $k \in \mathbb{N}$

- **Syntaxbaum:** Sei $G = (\Sigma, V, S, P)$ kontextfreie Grammatik und B geordneter Baum mit den Eigenschaften

  - Die Wurzel von B ist mit S markiert
  - Innere Knoten sind mit Variablen markiert, Blätter mit Buchstaben aus Σ

- Wenn der Knoten v mit der Variablen A markiert ist, und wenn die Kinder die Markierungen $v_1, \ldots, v_s$ tragenm dann ist $A \to v_1 \ldots v_s$ eine Produktion von G.

B ist ein Syntaxbaum des Wortes w, das durhc Konkatenieren der Markierungen der Blätter B (von links nach rechts) entsteht.
Grammatiken, die für jedes abgeleitete Wort genau einen Syntaxbaum besitzen, erlauben relativ einfache Lösungen des Compiler/Wort-Problems:

- G heißt **mehrdeutig** $\Leftrightarrow \exists w \in L(G)$ mit zwei verschiedenen Syntaxbäumen. Ansonsten heißt G **eindeutig**.
- Sprache L heißt **eindeutig**, wenn es eine eindeutige Grammatik G gibt mit $L = L(G)$. Ansonsten heißt L **inhärent eindeutig**.

Syntaxgraphen existieren für beliebige Grammatiken. Kontextfreie Grammatiken ermöglichen die Einschränkung auf Syntaxbäume, und reguläre Grammatiken kommen sogar mit Syntaxlisten aus (ein Hinweis, warum das Syntaxanalyseproblem für kontextfreie Grammatiken effizient lösbar ist).

- **Chomsky-Normalform:** Eine Grammatik ist in CNF, wenn alle Produktionen die Form $A \to BC$ oder $A \to a$ besitzen (mit $A, B, C \in V, a \in \Sigma$).
Wenn L kontextfrei ist mit $\varepsilon \notin L$, dann gibt es eine Grammtik G in CNF mit $L = L(G)$. Man kann sogar effizient zu einer kontextfreien Grammatik eine Grammatik in CNF konstruieren ($\to$Algorithmus; lediglich die Eliminierung der Kettenregeln kann hier zu Problemen führen, was eine Umfromung in Linearzeit verhindert).

  Kettenregeln: Ableitungen vom Typ $A \to B$. Können den Sytaxbaum sehr groß machen und sind deshalb verboten.

  Größe der Syntaxbäume: Das Worte der Länge n in höchstens $2n - 1$ Schritten abgeleitet werden könnenist die Größe der Syntaxbäume linear.

  Innere Knoten: Die Syntaxbäume haben für Worte der Länge n genau $n - 1$ innere Knoten und n Blätter, wobei jedes Blatt noch einen Vater mit nur diesem Blatt als Kind hat).

- **Cocke-Younger-Kasami:** Sei G in CNF, dann kann in Zeit $O(|\text{Produktionen von G}| \cdot |w|^3)$ entschieden werden, ob $w \in L(G)$. (Beweis mit dynamischen Programmieren: Bottom-up Ansatz, wobei für jedes Teilwort $w_i \ldots w_j$ bestimmt wird, aus welchem P_{ij} es sich ableiten läßt; da alle Teilprobleme gelöst werden müssen, bis P_{1n} ausgerechnet wurde, beträgt die Laufzeit $O(\binom{n}{2} + n)$; für jedes Teilproblem benötigen wir wieder $n \cdot |\text{Anzahl Produktionen}|$)).
Achtung: Die Laufzeit ist *immer* kubisch, da die Größe der Grammatik immer nur als linearer Faktor in die Laufzeit eingeht, während der Exponent entscheidend durch die Anzahl der Variablen auf der rechten Seite der Ableitungen bestimmt wird!

- **Pumping-Lemma für kontextfreie Sprachen:** Sei L kontextfreie Sprache, $\exists$Pumpingkonstante n $\forall z \in L$ mit $|Z| \geq n$ $\exists$Zerlegung $z = uvwxy$ mit Eigenschaft

  - $|vwx| \leq n$ und $|vx \geq 1$
  - $\forall i \geq 0 : uv^i wx^i y \in L$

  Also gilt:

  1. L_1, L_2 kontexfrei $\Rightarrow$ $L_1 \cup L_2$, $L_1 \circ L_2$ und L_1^* kontextfrei
  2. Es gibt kontextfreie Sprachen L_1 und L_2, so daß $L_1 \cap L_2$ nicht kontextfrei ist.
  3. Es gibt eine kontextfreie Sprachen L, so daß $\overline{L}$ nicht kontexfrei ist.

  Formal folgt also aus der Regularität von L:

$$\exists N \in \mathbb{N} \forall z \in L, |z| \geq N \exists \text{Zerlegung } z = uvwxy, |vwx| \leq N, |vx| \geq 1 \forall i \geq 0 : uv^i wx^i y \in L.$$

Zum Nachweis der nicht-Regularität müssen wir also die Umkehrung zeigen:

$$\forall N \in \mathbb{N} \exists z \in L, |z| \geq N \forall \text{Zerlegung } z = uvwxy, |vwx| \leq N, |vx| \geq 1 \exists i \geq 0 : uv^i wx^i y \notin L$$

- **Ogden's Lemma:** Sei L kontextfreie Sprache, dann $\exists$Pumpingkonstante n, so daß $\forall z \in L$ mit $|z| \geq n$ mit mindestens n markierten Buchstaben eine Zerlegung $z = uvwxy$ besitzt mit

  - höchstens n Buchstaben sind in vwx markiert
  - mindestens ein Buchstabe ist in vx markiert
  - $\forall i \geq 0 : uv^i wx^i y \in L$

  Das Pumping-Lemma folgt aus Ogden's Lemma, wenn wir alle Buchstaben von z markieren. (Beweis: Baum mit $n = 2^{|V|+1}$ markierten Blättern und also einem Weg der Länge mindestens $|V| + 1$, der immer die Abzweigung mit den meisten Blättern nimmt).
  Beispiel einer Grammatik, die so als nicht kontextfrei bewiesen werden kann:

$$\{d^r a^x b^y c^z | r = 0 \text{ oder } x = y = z\}$$

- **Greibach-Normalform:** Sei $G = (\Sigma, V, S, P)$ kontextfrei und alle Produktionen der Form $A \to a\alpha$ (für $A \in V, a \in \Sigma$ und $\alpha \in V^*$).
  Wenn G in CNF ist, dann gibt es eine Grammatik $G^* = (\Sigma, V^*, S^*, P^*)$ in GNF mit $L(G) = L(G^*)$ und $|P^*| = O(|P|^3)$.

- **Kellerautomaten:** Maschinenmodell, das genau die Klasse der kontextfreien Sprachen akzeptiert. Dabei können **deterministisch kontextfreie Sprachen** von einem deterministischen Kellerautomaten akzeptiert werden (für diese Sprachen gibt es sehr schnelle Algorithmen, die das Compiler-Problem lösen).

- **Nichtdeterministischer Kellerautomat (PDA):** $A = (Q, \Sigma, \Theta, q_0, Z_0 \in \Theta, \delta, F)$ mit Z_0 als anfänglicher Stackinhalt und $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Theta \to P(Q \times \Theta^*)$.
  Eine *Konfiguration* von A (für Eingabe $w \in \Sigma^*$) ist $(q, w'\alpha)$ mit $q \in Q, w' \in \Sigma^*$ und $\alpha \in \Theta^*$, wobei w' ein Suffix von w und α der Stackinhalt ist.
  Akzeptanz durch leeren Stack: $w = w_0 \dots w_n$, wenn eine Konfigurationsfolge

$$(q_0, w_1 \dots w_n, Z_0), \dots, (q, \varepsilon, \varepsilon)$$

  existiert. Der Keller darf also bis auf den letzten Schritt zu keinem Zeitpunkt leer sein.
  Akzeptanz durch akzeptierende Zustände: A akzeptiert w, wenn es eine Konfigurationsfolge $(q_0, w_1 \dots w_n, Z_0), \dots, (q, \varepsilon, \alpha)$ gibt mit $\alpha \in \Theta^*$ und $q \in F$ - der Keller muß auch diesmal leer sein.
  Wenn G in GNF ist, dann gibt es einen PDA A mit $L(G) = L(A)$ und A akzeptiert mit leerem Stack.
  Wird L von einem PDA A (mit leerem Stack) akzeptiert, dann gibt es eine kontextfreie Grammatik G mit $L(G) = L(A)$. Die Klasse der kontextfreien Sprachen entspricht also genau der Menge der Sprache, die von einem PDA mit leerem Stack erkannt werden.
  Wenn A_1 mit Zuständen akzeptiert, dann gibt es einen äquivalenten PDA A_2, der mit leerem Stack akzeptiert und umgekehrt.

- **Deterministischer Kellerautomat (DPDA):** Für jedes Tripel (q, a, z) mit $q \in Q, a \in \Sigma$ und $z \in \Theta$ gilt: $|\delta(q, a, z)| + |\delta(q.\varepsilon, z)| \leq 1$ und a mit Zuständen akzeptiert.
  L heißt **deterministisch kontextfrei**, wenn L von einem deterministischen Kellerautomaten erkannt wird (deterministisch kontextfreien Sprachen$\subseteq$kontextfreien Sprachen, da: DPA$\subseteq$DPDA).
  Akzeptanz durch Zustände ist bei DPDA's wesentlich mächtiger als die akzeptanz durch leeren Stack:

$$L_D^{Stack} = \{L \subseteq \Sigma^* | \exists \text{DPDA A, A akzeptiert L mit leerem Stack}\}$$

$$L_D^F = \{L \subseteq \Sigma^* | \exists \text{DPDA A, A akzeptiert L mit akzeptierendem Zustand}\}$$

Es gilt: $L_D^{Stack} \subset L_D^F$
Wenn L deterministisch kontextfrei ist, dann auch $\overline{L}$. (Konstruktion mittels $Q \times \{1,2,3\}$)
Es gibt det. kontexfreie Sprachen L_1, L_2, so daß $L_1 \cap L_2$ nicht einmal kontextfrei ist. ($L = \{a^n b^n c^n | n \geq 0\}$)
Die Menge der det. kontexfreien Sprachen ist unter der Vereinigung nicht abgeschlossen (Beweis mit Vereinigung von $\{a^x b^y c^z | x \neq y\}$) und $\{a^x b^y c^z | y \neq z\}$.
Die Menge der det. kontexfreien Sprachen ist eine echte Untermenge der kontextfreien Sprachen. Insbesondere ist $\{a^x b^y c^z | x \neq y \text{ oder } y \neq z\}$ kontextfrei, aber nicht deterministisch kontextfrei.
Außerdem gilt, daß

- Jede det. kontextfreie Sprache eindeutig ist
- $\{ww^{reverse} | w \in \Sigma^*\}$ eindeutig, aber nicht det. kontextfrei ist
- ein stets haltender DPDA auf Eingaben der Länge n ind Zeit $O(n)$ läuft. Das Wortproblem ist also für deterministisch kontextfreie Sprachen in Linearzeit lösbar.

- Sprachklassen und Ihre Charakterisierung: Übersicht:

Sprache	Charakterisierung
RA:	rekursiv aufzählbar, Turingmaschinen, die nur beim Akzeptieren halten müssen, Comsky-0 Grammatiken
REK:	rekursiv, stets haltende Turingmaschinen, keine Charakterisierung durch Grammatiken
KS:	kontextsensitiv, nichtdeterministische, linear platzbeschränkte Turingmaschinen, Chomsky-1, kontextsensitive oder monotone Grammatiken
DKS:	deterministisch kontextsensitiv, deterministische, linear platzbeschränkte Turingmaschinen
NP:	nichtdeterministische, polynomiell zeitbeschränkte Turingmaschinen
P:	deterministische, polynomiell zeitbeschränkte Turingmaschinen
KF:	kontextfrei, (nichtdeterministische) Kellerautomaten, Chomsky-2 oder kontextfreie Grammatiken
EKF:	eindeutig kontextfrei, eindeutige Grammatiken
DKF:	deterministisch kontextfrei, deterministische Kellerautomaten, $LR(k)$-Grammatiken für beliebiges, konstantes k
REG:	regulär, endliche Automaten, Chomsky-3, rechtslineare oder reguläre Grammatiken, reguläre Ausdrücke

Diese Sprachklassen stehen in folgenden Beziehungen zueinander:

$$REG \subsetneq DKF \subseteq EKF \subsetneq KF \begin{array}{l} \subsetneq DKS \\ \subsetneq P \end{array} \begin{array}{l} \subseteq KS \\ \subseteq NP \end{array} \begin{array}{l} \subseteq \\ \subseteq \end{array} REK \subsetneq RA$$

- Fragen:
- Definition?
- Spiel mit Odgen's Lemma
- Beispiele: kontextfreie Sprachen?
- Chomsky Grammatik
- Chomsky Normalform?
- Definition und Beweis: Greibach Normalform
- PDA
- Kellerautomat
- CYK-Algorithmus (Genaue Erklärung, wie sieht $v_{i,j}$ aus?)

- Warum profitiert der CYK-Alg. von dynamischer Programmierung?
- NFA→DFA in $2^{|Q|}$
- Beweis: Blow-up beim Übergang NFA→DFA?
- Beispiel: Potenzmengenkonstruktion, exponentielles Blowing
- Beweis: $a^n b^m c^{n+m}$ ist kontextfrei
- Gibt es kontextfreie Sprachen, die NP-vollständig sind? -Nein, wegen CYK
- Beziehungen zwischen rek. aufz. $\Leftrightarrow$ Typ 0, nicht det. TM lin. Platz $\Leftrightarrow$ Typ 1
- Ist $a^n b^n c^m$ kontextfrei?
- Wofür NFA's? (einfachere Beschreibung als DFA, weniger Zst.: NFA hat $|Q|$ und DFA $2^{|Q|}$ Zustände)
- Beispiel für NFA Zustandsvorteil: $\{0,1\}^* 1 \{0,1\}^k$ mit k fest (DFA benötigt nur Zeit $k+1$)
- Äquivalenz NDFA, DFA polynomiell?
- Wie beweisen: Sprache nichtdet. und kontextfrei?
- Definition: p-space
- Beispiel: Inhärent mehrdeutig?
- $L_{G_1} \cap L_{G_2} = \emptyset \rightarrow$ PKP def.